In the previous chapters we have given the foundations for the design of a new cognitive agent-based computational social simulation model. In this chapter, we will explain the design of a new cognitive architecture RBot, an object-oriented architecture that can easily be plugged-in into a multi-agent platform. In this chapter, we will design such a platform or task environment as well; our so-called Multi-RBot System (MRS).

However, we will first express again the need for such a new actor\(^1\) architecture. Hence, the first question for the reader should be: why developing another multi-actor simulation model and why not using existing software toolkits that give the possibility to model actors?

Although the problem of finding multi-agent toolkits is no problem (there are many available\(^2\)), many agent toolkits are developed with specific areas of research in mind. Such toolkits model for example social phenomena and are focused on the overall behaviour of populations (large populations), waiting queues and bottleneck studies. Or they focus on the intelligent agent itself and when that is the case, they focus only on certain aspects of the agent. Even when they focus on many aspects of the individual interacting with the environment, the assumptions of the model are most of the time not built upon findings in cognitive psychology (such as ACT-R does). Therefore, an initiative was taken to develop a new cognitive architecture from scratch, of course with other existing agent/actor technologies in mind.

This chapter will follow a structure that starts with an overview of the requirements of the architecture and these requirements will guide the design-process of our architecture. Next, a section will follow that elaborates the gen-

---

\(^1\)In this chapter we will use the term actor, because we are of the opinion that our agent is more autonomous and "similar" to a human actor than for example a web-agent.

eral architectural design of the complete system, i.e. what architectural patterns are applied. Following up this section, we apply a bottom-up approach in designing the system. We will first start a design process of a single actor and then go into details of how we will unite multiple actors into a multi-actor system.

The chapter will end with a short discussion that functions as an introduction to the next chapter. That chapter will discuss demonstrative experiments and the configuration of the actors and the task environment in detail.

5.1 General Requirements and Overview of the Architecture

In this section, we will first give the general requirements that are necessary for our architecture. Next, we will give a short explanation of patterns and then choose some of those patterns for the general design of our architecture.

The general requirements depend on the research questions we have posed in chapter 1. We will make use of the research questions (1—1.1 until 1.3) and the chapters about MAS, the social and the cognitive actor that deliver detailed requirements for the design of our RBot architecture. The research questions express three general requirements that the newly designed architecture should fulfil:

1. Research question 1.1 indicates that the system should support social behaviour and that such behaviour should be explained in terms of individuals and their interrelations. Distributed systems that are applied in DAI are suitable candidates for modelling such a system. The latest development in the area of distributed systems is the Multi-Agent System (see chapter 2), which is an appropriate methodology for modelling entities that interact with each other and the environment.

2. Chapter 3 answered research question 1.2 and discussed what was necessary for an actor in a MAS to exhibit social behaviour. The discussion made clear that the social construct is a (shared) unit of knowledge that can guide social behaviour. The social construct as a representation in the mind of the actor can be expressed as a data structure in the mind of the actor. Therefore, an actor needs a mind where representations can find their place.

3. Research question 1.3 addresses this issue, and more specifically, what kind of actor can handle representations, signs and social constructs. Chapter 4 explained the cognitive actor and stated that the ACT-R architecture can very well serve as an architecture for our cognitive plausible actor. Therefore, the requirement of our cognitive architecture is to inherit the model, the characteristics and the cognitive mechanisms of ACT-R in order to design a cognitive plausible actor.

The design of an architecture needs a methodology that supports proper design. Patterns provide solutions and methods for solving problems that often reoccur. In this chapter, we will adopt patterns and apply them for finding an
appropriate design that suffices our requirements brought forward by our research questions. In the next section, we will first explain shortly what patterns are and with help of those patterns, we will create an abstract design of our cognitive agent-based social simulation system.

### 5.1.1 Patterns

Patterns are reusable solutions for problems that occur during software development. Experienced designers (or programmers) find themselves often in situations or problems that have occurred before. In our case of designing a MAS, patterns can provide solutions for the design of our architecture. But what is exactly a pattern?

Christopher Alexander defined a pattern as follows:

> Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution... [It] describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over... (Alexander, Ishikawa, Silverstein, Jacobson, & Angel, 1977, p. x)

There are many classes of software patterns, i.e. architectural patterns, analysis patterns, design patterns, creational patterns, structural patterns, and so on, from which design patterns are the most well known (cf. Gamma, Helm, Johnson, & Vlissides, 1995). Three conceptual levels of patterns can be distinguished by categorising patterns into architectural patterns, design patterns and idioms (or coding patterns) (Buschmann, Meunier, Rohnert, Sommerlad, & Stal, 1996).

- An **Architectural Pattern** expresses a fundamental structural organization or schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.

- A **Design Pattern** provides a scheme for refining the components of a software system, or the relationships between them. It describes a commonly recurring structure of communicating components that solves a general design problem within a particular context.

- An **Idiom** is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

In this chapter, we want to elaborate the architectural aspects and not a complete technical description of the architecture. We will adopt architectural patterns to describe the design of the architecture. The architectural patterns can be classified according to architectural views:

> An **Architectural View** is a representation of a system from the perspective of a related set of concerns (e.g. a concern in a distributed
system is how the software components are allocated to network nodes. This representation is comprised of a set of system elements and the relationships associated with them (Clements, Bachmann, Bass, Garlan, Ivers, Little, Nord, & Stafford, 2002). An Architectural Pattern, on the other hand, defines types of elements and relationships that work together in order to solve a particular problem from some perspective. (Zdun & Avgeniou, 2005, p. 4)

In designing our architecture in the next section, we will first select the views that match the concern. According to these views, we will select the architectural patterns that provide the best solution for our problem.

5.1.2 Overview of the architecture

The overview of the architecture has been designed based on a combination of a set of general architectural patterns. We will follow the design of a MAS (requirement 1), which requires that the components of the architecture should have a high degree of autonomy. The Component Interaction View (CIV) (Zdun & Avgeniou, 2005, p. 28) contains individual components that interact with each other but retain their autonomy, i.e. they merely exchange data but do not directly control each other. Interaction can be performed synchronously or asynchronously and can be message-based or through direct calls. These components can be distributed. Therefore, the CIV is closely connected to the Distributed Communication View (DCV). The CIV and the DCV are suitable to be applied for designing a MAS, because of their characteristics of autonomy and distribution, respectively. The overview\(^3\) of patterns associated with CIV and DCV is displayed in figure 5.1.

In most multi-agent simulation systems, autonomous actors 'live' in a task environment, which is a separate process that is inaccessible but can be influenced by the actors. Hence, in the design of our architecture, we have made the decision to separate the task environment and the actors. Because the task environment (supplier of the environment) and the couplings with the actors (clients: consumers of the environment) are known at design time, the Explicit Invocation\(^4\) pattern is a general architectural pattern that can be applied for our architecture. Within the Explicit Invocation we can make a decision between the Client-Server (CS) pattern and the Peer-to-Peer (P2P) pattern. The CS pattern consists of a client and a server, two independent components that are not equal to each other: one, the client, is requesting a service from the other, the server. On the other hand, the P2P pattern is somewhat similar to the CS pattern, however all components are equal and can provide as well as consume services. The pattern applied in our design is the Client-Server pattern, i.e. the components client and server are not equal; the server is the task environment and the actors are the clients.

\(^3\)For a complete view, see Zdun and Avgeniou (2005).
\(^4\)In the Implicit Invocation pattern, the invocation is not performed explicitly from client to supplier, but indirectly through a special mechanism (Zdun & Avgeniou, 2005, p. 31).
5.1. General Requirements and Overview of the Architecture

Apart from a server that delivers the task environment, we also want to implement a backend server that has the function of storing simulation data that are produced by the actors. In that case, results can easily be interpreted after a simulation did run successfully. Hence, when the simulation runs, the actors store their experiences in the backend server. After a run of a simulation, the server changes its functionality and becomes a client that retrieves data from the backend server and presents data or graphs to the user of the simulation program. The combined design delivers a 3-tier-architecture (called Multi-RBot System or MRS) that is displayed in the following figure.

The 3 components that can be discerned are the following:

1. The actor: the actor is an autonomous cognitive plausible actor consisting of a production system called RBot which is similar to ACT-R and is able to register itself with the task environment and thereby able to perceive the physical environment and interact with other actors. The actor is the most complex component of the complete architecture and will be discussed in section 5.2.

2. The backend server: the function of the backend server is to store experiences that each actor encounters during its lifetime in a simulation run. This varies from sub-symbolic data (activation), symbolic data (chunks) and physical data, such as place, time etcetera. The database functionality will be discussed in section 5.3.

3. Mixed tier (server-client): a component that serves three functions: the first function, in the function of server, is taking care of control, registration, synchronisation and communication between actors. The second function is presenting a virtual (simulated) world and supplying a task environment—a physical and communication environment—for actors to
live in, i.e. it enables agents to perceive each other and communicate with each other. The user can configure the input parameters of the task environment (e.g. number of agents allowed in the simulation) and the process (e.g. start, pause and resume the simulation) of the simulation. The third function, as a client, is concerned with the presentation of output data generated by the simulation that has accumulated in the backend server, i.e. as a client, the component is able to process output data in order to present them in graphs or output-files to the end-user, ready for interpretation. We will elaborate the mixed-tier component in section 5.4.

The Client-Server pattern seems to be an appropriate pattern for our architecture. However, in a Client-Server architecture, the client has a direct connection with the server and when it makes a request to the server, the server is a passive component that only responds to its environment and is not pro-active. In our architecture, we require a more active component for the server than the CS pattern describes, i.e. the server controls synchronisation between the agents by sending time-step events to the agents. Therefore, we adopt the Publish-Subscribe (PS) pattern (see figure 5.1) to complement our architecture.

[The Publish-Subscribe pattern] allows event consumers (subscribers/[actors]) to register for events, and event producers to publish (raise) specific events that reach a specified number of consumers. The [PS] mechanism is triggered by the event producers and automatically executes a callback-operation to the event consumers. The mechanism thus takes care of decoupling producers and consumers by transmitting events between them. (Zdun & Avgeriou, 2005, p. 35)
5.1. General Requirements and Overview of the Architecture

The PS pattern allows for more autonomy for the agents by a decoupling mechanism, i.e. the PS pattern creates a “bridge between the asynchronous network events and the synchronous processing model of the server” (Zdun & Avgi, 2005, p. 36). In our architecture, we want actors to subscribe or register themselves with the server, which is a publisher of task environment events. These events can vary from informing of time-progressive, or events that happen in the environment, e.g. change of objects in the field are announced to actors that are in the vicinity of these objects.

Another issue we want to discuss in the overview of the architecture is the communication environment, i.e. how do actors communicate (publish and consume events or messages) with each other with help of the server. The most common communication patterns in the Distributed Communication View are the Remote Procedure Call (RPC) and the Message Queueing (MQ) (Zdun & Avgi, 2005, pp. 38–39).

The choice for our architecture is to prefer Message Queueing above RPC. The main reason is that we want to have a communication channel that is reliable even when neither the network nor the receiver is reliable. “The simplicity of RPC’s is that they’re synchronous; a call happens all at once, the caller blocking while the receiver processes. But this is also the shortcoming of an RPC; if anything goes wrong, the whole thing fails” (Woolf & Brown, 2002, p. 4). Therefore, we “[u]se messaging to make intra-program communication reliable, even when the network and the receiver program cannot be relied upon” (Woolf & Brown, 2002, p. 4).

The last aspect we want to discuss is the Interaction Decoupling View that “shows how the interactions in a system can be decoupled, for instance, how to decouple user interface logic from application logic and data” (Zdun & Avgi, 2005, p. 24). In our 3-Tier architecture, all three components (actor, server and backend) are equipped with a separation of model, view and control, i.e. a Model-View-Controller (MVC) pattern. The MVC divides the system into three different parts:

[A] Model that encapsulates some application data and the logic that manipulates that data, independently of the user interfaces; one or multiple Views that display a specific portion of the data to the user; a Controller associated with each View that receives user input and translates it into a request to the Model. View and Controllers constitute the user interface. The users interact strictly through the Views and their Controllers, independently of the Model, which in turn notifies all different user interfaces about updates. (Zdun & Avgi, 2005, p. 25)

Figure 5.3 shows an example of a Model-View-Controller pattern.

The main advantage of separating model from view and controller is that one of these three components can change (e.g. adding/changing a view) without affecting the structure of the others.

A similar advantage is the decoupling of actors and environment. In the first place, we can run a simulation with a task environment, i.e. the actor ‘lives’ in a
task environment and communicates with other actors—the MRS-mode (Multi-RBot System). On the other hand, it is possible to run a single actor and perform a stand-alone simulation. The stand-alone simulation is similar to the ACT-R simulation.

In the next chapter, the first experiment shows a RBot stand-alone simulation in which the actor performs a cognitive arithmetic calculation of adding numbers. The second and third experiment are configurations of two RBot actors that are situated in a task environment (MRS-mode). The remainder of the chapter will start with the design of the cognitive plausible actor architecture, i.e. the stand-alone RBot actor. Next we will discuss the other components—the database and the task environment—that together with RBot form the MRS (Multi-RBot System).

5.2 The cognitive actor architecture

In this section we want to discuss the design of the cognitive architecture. The cognitive plausible actor/cognitive architecture is the most elaborate component of our complete software architecture. The architecture is autonomous, in the sense that it can function independently and can be integrated with any Multi-Agent System, varying from an environment with web-based agents, a robotics environment or a simulation environment as in our case. Apart from autonomy, we want to state that a cognitive actor requires a physical symbol system in order to exhibit intelligent action (Newell, 1980). Such a physical symbol system can be implemented with help of a production system.

From a system architecture perspective, production systems like our architecture RBot, ACT-R and SOAR are partly interpreters, and more specifically: they are rule-based systems, which is a pattern with the following properties:

[A Rule-Based System] consists of mainly three things: facts, rules, and an engine that acts on them. Rules represent knowledge in form of a condition and associated actions. Facts represent data. A
5.2. The cognitive actor architecture

Rule-Based System applies its rules to the known facts. The actions of a rule might assert new facts, which, in turn trigger other rules. (Zdun & Avgeriou, 2005, p. 24)

RBot is a rule-based system, however, just like ACT-R, it is more than just a rule-based system: it is a cognitive architecture and comprises among the rule-based system cognitive mechanisms that are based on cognitive theories as explained in chapter 4.

The structure of this section will discuss the components of the actor architecture RBot, i.e. the memory model and components: the declarative, procedural and working memory (goal-list and goal stack), the perception component, the physical memory (awareness) component, the message memory component, the social construct memory component and the cognitive engine.

The actor’s architecture (figure 5.4) resembles the model of ACT-R version 4 (Anderson & Lebiere, 1998) and version 5 (Anderson et al., 2004). The model is the result from research in software development during which we developed a memory model and a production system based on ACT-R, i.e. the RBot project (cf. Roest, 2004).

If we compare the model of RBot with the model of ACT-R in the previous chapter, then there are a couple of differences. First the access to the memory (memorymap), discussed in the next section, is different in comparison with ACT-R. Next, the function of the goal stack, the cognitive engine\(^5\) (matching, conflict resolution and execution), the declarative, procedural and perception memorymap are similar to the model of ACT-R which is discussed in the previous chapter. The difference with ACT-R is the addition of a goal feeder, the social construct memorymap, the message memorymap and the physical memorymap. The function of the goal feeder is to store goals that need to be processed, i.e. it is possible to equip the actor with a plan of goals and these goals are delivered one by one to the goal stack for the cognitive engine to be solved. The motor part of RBot consists of a physical memorymap that maintains the current position and time of the actor (physical awareness) and the message memorymap that regulates communication with the outside world. The social construct memorymap is a new feature that maintains (social) knowledge about the role the actor plays in the social environment with other actors. The social construct allows the actor to react instantaneously to social events and enables the actor to respond to perception and messages from other actors received from the outside world. These interactions can for instance adjust sub-symbolic properties, such as the utility of procedures and thereby influencing the execution of procedures. The remainder of the section will explain the different components in detail.

The model and the memory of the actor are organised in a similar structure as ACT-R, however the way the model is implemented in JAVA is quite different. Not only the language chosen to implement the model is different, also the design of the memory and its components is quite different. Because the model

\(^5\) Although the human brain does not contain a cognitive engine, we refer in this dissertation to it as a reasoning unit that controls the operations on symbols or representations in the mind of the actor.

167
is implemented from theory, and not from the original LISP code, the model may show differences. However, the way the model can be used in practice is similar to ACT-R.

Building up the model from scratch towards a multi-actor simulation model needs a clear explanation about the implementation of the model and its environment.

The cognitive actor architecture (the client) can be divided in three main components: the memory model, controller and handlers, and the user interface, similar to the MVC pattern as discussed in the previous section. The memory model (next section) together with the controller and handlers (section 5.2.2) form the core of the RBot architecture and will be explained in detail.

Besides that, every actor that is part of the multi-actor simulation has its private user interface. This interface (section 5.2.3) allows for configuration and control of the single actor. The user interface of the task environment is a different user interface and is part of the mixed-tier, which we will discuss in section 5.4.

5.2.1 The Memory

The model of the memory arranges information inside the actor’s mind. The importance of this can be seen when we for example see the arrangement of the cupboards and the refrigerator in a kitchen. For instance, the moment we feel the need for a cold drink, we tend to go to the refrigerator to get it, i.e. we know from the refrigerator that it is capable of holding drinks and keep them cold; and in our mind, we label the refrigerator as having this capability. Similarly, in the memory of an actor, the indexing and labelling of information (cf. tagging; Holland, 1995) are necessary for the ordering of information and retrieval of information at low cost.

The memory model manages the state and the access of the memory. In a
production system, like RBot, productions and facts are stored in memory in order to be retrieved, changed or added, which is caused by state changes of the actor. The results of these state changes have to be stored in an orderly fashion, allowing easy access and management of the memory.

The memory of RBot consists of components: *chunks*, the generic components of storing information or knowledge in memory that are linked by *links* in a network-like structure forming a graph structure. Chunks contain atomic components, namely slots that can contain objects or data. Besides the graph structure, the formatting (the number and type of slots in a chunk) is defined by its *chunktype* (*its tag*). Hence, apart from creating structure in memory, it is possible to classify chunks as well, based on their chunktype.

RBot as well as ACT-R model separate parts in memory to store information (e.g. procedural and declarative memory). Secondly, in both architectures, the components themselves are classified as well (tagging: chunktype). Thirdly, ACT-R and RBot give importance to chunks by ranking them based on activation level (explained later on). Besides some overlap in the structure between RBot and ACT-R, RBot has an explicit graph structure⁶ (comparable to the production-graph structure of SOAR) that is a set of vertices (chunks) connected by edges (links). In this way, a network of chunks connected by links can form a semantic network. In RBot, the memory holds a collection of memory maps. A memorymap, for instance the declarative memory, is a map that holds a graph of chunks and links. The organizing of graphs in different maps gives the possibility to assign different functionality to those maps (for instance, perception memorymap connects to functions that relate to perception). The way this is done in RBot is slightly different and will be explained in the next section.

### 5.2.1.1 Memorymap, buffer and stack

The design of the memory follows closely the design of ACT-R in describing the components. We also define three main memory modules: the goal stack, declarative and procedural memory. In RBot, these main memory modules are special cases of a generalised memory organisation based on memory maps.

A *memorymap* is a container of memory components that gives the actor access to these memory components. Whereas ACT-R applies a functional division of the memory in modules, the memorymap in RBot is a better and more flexible approach to structure the memory. The memorymap creates access to chunks in memory by defining entrance chunks, see figure 5.7.

Chunks can be defined as part of different memory maps in memory. For example, a chunk can be part of the declarative memory, but at the same time can also be part of the procedural memory, without being duplicated in the memory. This is possible because the memorymap holds references to objects in memory and forms an object-oriented memory structure. In other words, chunks are separated in different maps, but they are not that strict separated that they cannot be part of several maps.

⁶Graph structures (implicit graphs/soft links) are also possible in ACT-R by connecting slots in chunks to other chunks.
RBot defines three types of memory maps (figure 5.5): the *memorymap*, the *buffer* and the *stack*. The memorymap is the super container of the memory that can theoretically contain an infinite number of chunks. Buffer and stack are both based on the memorymap and inherit properties from the memorymap.\(^7\)

![Diagram of memory types](image)

Figure 5.5: Three types of memorymaps.

The buffer has a predefined size and behaves like a FIFO (First In First Out) buffer that allows a buffer to store only a maximum amount of chunks at any time. For example, the visual-buffer could be defined as containing a maximum number of elements: when more items are stored in the visual-buffer, the buffer removes items that have stayed in the buffer the longest.

The stack on the other hand acts like a LIFO (Last In First Out) buffer with infinite size and points at the latest addition of a chunk, i.e. the chunk that is added (pushed) on top of the stack will be removed when there is a remove (pop) request made to get a chunk from the stack.

The generic methods of accessing and controlling a memorymap are adding chunks, retrieving chunks, and searching for chunks based on their properties, e.g. a certain defined chunktype. Figure 5.6 shows an example of a buffer with a capacity of five chunks to demonstrate the way a memorymap (buffer) functions.

In this case the buffer is full, adding another chunk will make the fifth chunk of chunktype X automatically be removed. And for example, when we want to retrieve chunks of this memorymap, we can retrieve all the chunks or chunks that are based on a certain defined chunktype (its label). Thus, the function of the memorymap is: organising memory components and giving access to them according to the properties of the memorymap type (in this case a FIFO buffer).

In RBot, it is possible to define custom memorymaps and add them to memory. Some memorymaps are standardised and have a predefined function. These memorymaps are the following:

- Goal feeder memorymap: manages and schedules the supply of goals to the goal stack.

- Goal stack: the internal stack of the goalhandler that manages sub goaling.

---

\(^7\)The inheritance principle (Campione, Waalkens, & Huml, 2001, p. 52) is an object-oriented property. Hence the memorymap and its children are object-oriented classes.
5.2. The cognitive actor architecture

![Diagram of a buffer/memory map]

Figure 5.6: Example of a buffer/memory map.

- Declarative memory map: has the function of storing chunks that represent facts.
- Procedural memory map: stores the procedure chunks in memory.
- Social constructs memory map: storage of social constructs.
- Physical memory map: storage of physical properties of the actor.
- Message memory map: messages, in the form of chunks, destined for communication with the outside world are stored here.
- Perception memory map: messages of perception are stored in this map in the form of chunks.

Every memory map has the ability to store memory components such as chunks. These components are holders of information and describe states in memory and influence how state changes can take place. The design of these components will be discussed next.

### 5.2.1.2 Memory Components

The Memory Component is the super-class of all components (e.g., chunks, links) in the memory. Its main function is to administer the activation of the components and making them aware of time progress during the simulation. All components that extend the generic Memory Component have the properties of activation and time awareness at the so-called sub-symbolic level.

After creation of a memory component, the component (be it a chunk or link) keeps track of its presentations (the number of times the component has been harvested or merged by the engine), thereby storing the history of its presentations in order to be able to determine its activation value. The calculation
Chapter 5. RBot: Architecture and Design

and determination of the current activation of the memory component will be explained in the next section about controllers and handlers.

In this section, we will start with explaining the chunk, the link and the way those components can be structural and managed. Next, we will discuss the procedure, a component that consists of a structure of links and chunks and the formatting of chunks (XML) that enables the actor to translate memory components into computer files or messages that can be sent to other actors. We end this section with the social construct; a memory component comprised of a structure of links and chunks that gives the actor the capability of constructing social situations in its mind.

Memory Component :: Chunk

Chunks are the most important memory components in the architecture, they are the building blocks for most other memory components. With help of links (another memory component), chunks can be connected to other chunks and can form complex structures, comparable to molecular structures or neural networks. The chunk is a container that contains slots. A slot has a name and holds objects or data. With help of a classifier, the chunktype, the number and names of slots in a chunk are defined (tagged), comparable to the DTD\(^8\) of an XML document.

Chunktype Chunktypes are defined in main memory and only chunks that are valid according to a predefined chunktype are accepted as "understood" elements. To compare with XML documents: when XML documents are transferred between computers (or actors), they only can be parsed if they are following a defined format; only chunks that are defined by a known chunktype are parsed by the actor and stored in memory. New chunks of an unknown chunktype can only be absorbed by the actor's memory if the chunktype is given as well or if a separate mechanism exists for storing unknown chunks and allows for the generation of new chunktypes. Chunktypes give restrictions in the way information is stored, but give the ability for indexing and easier search of specific types of information stored in memory.

Memory Component :: Link\(^9\)

The link is a memory component that enables to connect two chunks, thus creating the possibility to construct a graph or network structure. The advantage of links is that they order memory by linking certain chunks that are semantically related to each other.

For instance, connect chunk5 (value 5) together with chunk7 (value 7) and chunk12 (value 12). In this case, an addition or subtraction of two chunks would result in a faster retrieval from memory because they are directly connected to

---

\(^8\)DTD (Document Type Definition) defines the legal building blocks of an XML document.

\(^9\)In the current RBot architecture, the activation levels of links are not used, but its availability can be used for future algorithms.

172
each other\textsuperscript{10}. In the definition of RBot, the link can contain two chunks only, while the chunk can be related to an unlimited number of links. The current implementation of the link is bidirectional, i.e., the moment a link is constructed, both chunks have a link to each other. With the help of the chunktype, chunks can already be sorted, but the addition of links between chunks gives the possibility of creating very complex structures such as trees, or other types of graph structures that order chunks even more, e.g., a semantic network.

**Memory Component :: Structure**

With help of links, chunks and a memorymap, it is possible to create ordered (complex) structures of chunks in a memorymap. Figure 5.7 shows an example of a created memory structure. The entrance chunks form the elements that are directly accessible from outside the memorymap and are explicitly added to the memorymap. All other elements can only be reached indirectly, by following a specific link towards them.

![MemoryMap Entrance Chunk Chunk Link](image)

**Figure 5.7:** Example of a memory structure.

The purpose of the entrance chunks is to create a separate access layer in the memorymap. The advantage of such a layer is that it creates faster access and sorting of chunks that are the most important.

The structure of components in the memorymap and the relations between its components can be explained with help of an UML\textsuperscript{11} diagram, see figure 5.8.

The class diagram shows that an actor has a memory that exists out of one or more memorymaps. Apart from that, the memory also stores the chunktypes that are known to the actor. The memorymap, as discussed before, can be a simple container (List) of chunks, a FIFO buffer or a LIFO stack. The ChunkType defines the structure of the Chunk. Both, the Chunk and ChunkType, are constructed out of Slot components. The Link and the Chunk are memory components that inherit properties and methods from the MemoryComponent, i.e. Chunk, Link and ProcedureChunk (explained later) inherit properties, such as

\textsuperscript{10}If the three are connected with many other chunks, then it can result in slower retrievals according to the fan-effect. See previous chapter and Anderson (1974).

the field ActivationValue and the method timeElapsed() from the MemoryComponent. The MemoryComponent has a listener that responds to time-based events generated by the simulation clock (the Rhythm).

With this set of memory components, it is possible to create complex graph structures, e.g., a semantically linked network structure. Consequently, a memory of memory maps with chunks, interconnected by links and holding slots formatted by the chunktype, can be defined as part of the actor.

**Memory Component :: Management**

In order to maintain the memory, RBot defines functions for managing memory: adding or creating, moving and removing, and searching for chunks and links. Chunks can be created by internal cognition or by encoding of signs perceived from the environment.

The chunk can be added to the memorymap in two ways: adding chunks directly to the memorymap as an entrance chunk or adding by linking the chunk to another chunk that is already part of a memorymap, i.e., in the case of procedure chunks, a new constraint can be added to a condition side by linking a new chunk to the condition Rule (explained in the next section). In the case of a link, it can be added by specifying the two chunks to which it needs to be attached.

Removing any memory component is exactly the opposite of adding a component and moving a component is a combination of first removing and then adding.

Indexing (classification by index, i.e. tagging/chunktype) memory gives the ability to create specific algorithms for memory-search because knowledge is
5.2. The cognitive actor architecture

classified by chunktype, linked to other chunks and member of a certain memorymap.

Algorithms to search knowledge can vary and many possibilities to search memory are possible. The singleton Algorithm class (discussed in the handler section) is a handler that handles all complex searches for information in memory and is separated from the memory components and memorymap. The separation of search-algorithms from memory objects has the advantage of easier and better maintenance and reusability.

Multiple references to chunks In RBot, there is no restriction in having double or more references to chunks; chunks can be member of many memorymaps at the same time. The memorymap is used to create different parts in memory that are designated a different functionality, i.e. the same chunks can become member of different memorymaps. In this case, the chunk is still unique, but there are more ways of reaching the chunk by reference in memory.

The remaining issues regarding management are more related to processes than to structure. Hence, we refer to the next section that elaborates controllers and handlers.

MemoryComponent :: Procedure, Rule, Demand and variable chunks\textsuperscript{12}

Both ACT-R and RBot distinguish different cognitive components regarding the declarative memory and the procedural memory. Whereas the declarative memory contains components that describe states and facts in memory, the procedural memory contains components that describes state changes.

The difference between ACT-R and RBot is the design of the software components. In ACT-R, a separate component is created for defining a procedure, whereas in RBot, the procedure is a small network of chunks that contain information about the procedure.

In RBot, a production or procedure is a small network with a procedure chunk as root, to which rule chunks are linked. A production normally exists out of two sides, a condition and an action side. The main function of the rule chunk is to make clear to which side constraints (or demands) belong and to which memorymap that condition or action refers to, e.g. as goal stack condition, declarative memory condition, goal action and so on.

The rule chunks are linked to demand chunks. When the demand (or pattern\textsuperscript{13}) chunk is connected to a condition rule chunk (that refers to a certain memorymap), then the demand chunk states to which pattern the chunks in the referenced memorymap have to be matched to. In the case of an action rule chunk, the demand chunk will be processed by the cognitive engine that can result in a change of the contents in the referenced memorymap.

The role of variables in a procedure is to allow for procedures containing patterns that match to more than only one situation. The influence of variables

\textsuperscript{12}During development, these names (rule, demand and so on) evolved. We see a production as a combination of condition-action rules. A production consists out of a set of condition and action rules that apply to that production.

\textsuperscript{13}A pattern chunk is a chunk that contains variables and describes which other chunks do match with this chunk (pattern-matching).
will be discussed in the section that discusses generalisation and specialisation of procedure chunks.

An additional feature of RBot compared to ACT-R is that RBot allows for variables in the name of a slot in a demand chunk. These variables are expressed in a separate DSN (Dynamic Slot Name) chunk, a holder of slotname variables that is linked to the demand chunks. The DSN chunk will also be explained in the section about generalisation and specialisation. The complete structure of the procedure is shown in figure 5.9.

![Figure 5.9: Structure of the procedure.](image)

The condition and action chunks are defined by the Rule-chunktype that exists out of three slots: the type (‘part’), modus and referenced memorymap. The type defines if the chunk is a condition or action rule chunk, the “modus”, expressed by a ‘-’, ‘+’ or ‘=’, determines what type of operation on the referenced memorymap, the third slot, is taking place. In the current implementation of RBot, there is only one condition modus possible, the ‘=’ sign, which stands for compare; comparing in the sense of a match between the condition of the procedure and the goal that is currently in focus or chunks that have to be retrieved. The action side on the other hand has more possibilities, the ‘-’ is the action of modifying, the ‘+’ for removing and the ‘*’ for adding chunks. The demand chunk defines the specific request to the condition or action chunk and determines what slots of what type of chunk the operation has effect on.

The advantage of splitting the procedure into different parts is shown when adding another procedure P2 that reacts on similar conditions, but has a different action A2 (see figure 5.9). Procedures can be interlinked when they possess similar conditions, actions or demand chunks. By creating linkages, memory can be saved and managing the procedures can be done more efficiently.

Inside a part (the condition or action part), the specific constraints or demands to the conditions and actions can be set. Procedures can only be activated

---

14We follow as much as possible the conventions of ACT-R.
when their condition side matches with the current state of the goal. Hence, the procedure always contains a demand chunk that refers to the goal stack, otherwise it will never respond to goals.

**Generalisation vs. specialisation of procedure chunks** In the previous chapter, we explained learning mechanisms at the symbolic level. Anderson and Lebiere (1998, p. 106) distinguished discrimination, generalization, composition and proceduralization. Procedures that are created by such a learning mechanism can either become more general or more specific applicable to the context in which they operate. We define two processes: generalisation (Anderson's generalization and proceduralization) and specialisation (Anderson's discrimination and composition). We can infer from these processes that there is a variation and change in the structure of procedures as a result from generalisation and specialisation, varying from more general towards more specific procedures.

General procedures are applicable to situations in which a procedure needs to solve more than solely one goal (or one specific retrieval). How general a procedure is, depends on the number of condition chunks and the number of variables that are available in the demand chunks and DNS chunks linked to those demand chunks. Fewer condition chunks, defines fewer requirements that are necessary to match the condition of the procedure, and therefore the procedure is more general applicable.

The strength of variables is that it allows for defining procedures that are able to react on more than one situation. Thus, the more variables defined in a condition-rule (e.g. the goal-condition) the more situations are applicable for the procedure to react on.

A variable is expressed by an 'Ξ' in front of the variable name, e.g. =num1 (similar to ACT-R). The variable gives the ability to transport a value from the condition side of a production to the action side. The more variables in the demand chunks at the condition side, the wider the range of goals the production can react on, the more generalised the procedure. Compare for example the following demand chunks\(^{15}\) that react on a goal of adding two numbers.

The goal chunk is of chunktype addition problem with two slots known (number1 == 2 and number2 == 3) and the value of the answer is unknown. The condition demand chunks are chunks that are to be matched with the goal chunk. Both chunks are responding to the goal chunk, however, demand chunk 1 has two variables and also responds to a goal chunk with for example number1 == 4. Demand chunk 2 is more specific and only responds in cases where number1 equals 2. Thus, demand chunk 2 is more specific than demand chunk 1.

The case of generalisation can go one step further by introducing more places in the procedure where variables can be substituted. The open production issue, discussed in the ACT-R version 6 Proposals (Bothell, Byrne, Lebiere, & Taatgen, 2003), suggests to create variables in the chunktype and the slot name (which

\(^{15}\)We apply XML (eXtensible Markup Language) notation for our chunks. See http://www.w3.org/XML.
could have some implications at the theory level of ACT-R). RBot does not implement the variable chunktype, but does implement variable slot names (DSN chunks) in procedures and can be seen as an important innovation. The reason of implementation is that during development of complex problems more generalisation in procedures proves to be necessary. The more generalisation, the more abstract the procedures, and the less procedures are necessary to describe a problem space. The following example shows a possibility of applying such a generalisation\(^{16}\).

The condition states that the production reacts to all goal chunks that have their slot name [number\(<N>\)] matching their data value \(N\), e.g. one chunk that matches is the following goal chunk.

With this additional property, it is possible to write procedures that are applicable for a larger domain of goals.

**Sub-symbolic level** The root chunk (e.g. P1 and P2) also functions as a holder for data at the sub-symbolic level. All sub-symbolic data is stored inside

---

\(^{16}\)The purpose of the example is only to show that it creates a more general applicable condition chunk when a slot name can hold variables, i.e. it is not directly applicable to solving an addition problem.
5.2. The cognitive actor architecture

slots that are defined by the procedure chunk type. It serves as the holder for efforts, successes and failures, and therefore represents the sub-symbolic level of procedures. This is done by purpose in order to configure easily new sub-symbolic data holders for flexible implementation extensions for procedures in future releases. More details about the sub-symbolic level are found in the next section about controllers and handlers.

Memory Component :: Formatting components in XML

In the discussion about the chunk type, the introduction of the DTD and XML was already mentioned. XML stands for Extensible Markup Language (Harold, 1999), derived from SGML (ISO 8879), and is a set of rules for defining semantic tags that break a document into parts and identify different parts of the document. XML is an industry-wide adopted standard, documented at the W3C (http://www.w3.org/XML), and is already applied in many software applications.

RBot applies XML to translate memory components and memory structures into messages or computer files. This structure gives the possibility for easy transfer between components inside the architecture, storage of components in documents and transfer between actors, or other software applications if necessary. The XML structures in the next table show examples of how a procedure and a goal chunk are modelled in RBot.

<table>
<thead>
<tr>
<th>Example of Procedure in XML</th>
<th>Example of GoalChunk in XML</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;procedurechunk name=&quot;StopCounting&quot;&gt;</td>
<td>&lt;goalchunk name=&quot;addition_problem&quot; activation=&quot;0.7&quot; type=&quot;addition_problem&quot;&gt;</td>
</tr>
<tr>
<td>&lt;condition&gt;</td>
<td>&lt;slot name=&quot;col1number1&quot; data=&quot;5&quot; /&gt;</td>
</tr>
<tr>
<td>&lt;rule nodus=&quot;&quot; memorymap=&quot;goalbuffer&quot;&gt;</td>
<td>&lt;slot name=&quot;col1number2&quot; data=&quot;7&quot; /&gt;</td>
</tr>
<tr>
<td>&lt;chunk name=&quot;goal&quot; type=&quot;countfrom&quot;&gt;</td>
<td>&lt;slot name=&quot;col2number1&quot; data=&quot;5&quot; /&gt;</td>
</tr>
<tr>
<td>&lt;slot name=&quot;Start&quot; data=&quot;num1&quot;/&gt;</td>
<td>&lt;slot name=&quot;col2number2&quot; data=&quot;7&quot; /&gt;</td>
</tr>
<tr>
<td>&lt;slot name=&quot;End&quot; data=&quot;num1&quot;/&gt;</td>
<td>&lt;slot name=&quot;col3number1&quot; data=&quot;1&quot; /&gt;</td>
</tr>
<tr>
<td>&lt;slot name=&quot;Step&quot; data=&quot;counting&quot;/&gt;</td>
<td>&lt;slot name=&quot;col3number2&quot; data=&quot;7&quot; /&gt;</td>
</tr>
<tr>
<td>&lt;/chunk&gt;</td>
<td>&lt;slot name=&quot;col4number1&quot; data=&quot;5&quot; /&gt;</td>
</tr>
<tr>
<td>&lt;/rule&gt;</td>
<td>&lt;slot name=&quot;col4number2&quot; data=&quot;7&quot; /&gt;</td>
</tr>
<tr>
<td>&lt;/action&gt;</td>
<td>&lt;slot name=&quot;carry&quot; data=&quot;start&quot; /&gt;</td>
</tr>
<tr>
<td>&lt;rule nodus=&quot;&quot; memorymap=&quot;goalbuffer&quot;&gt;</td>
<td>&lt;slot name=&quot;col4number1&quot; data=&quot;5&quot; /&gt;</td>
</tr>
<tr>
<td>&lt;chunk name=&quot;goal&quot; type=&quot;countfrom&quot;&gt;</td>
<td>&lt;slot name=&quot;col4number2&quot; data=&quot;7&quot; /&gt;</td>
</tr>
<tr>
<td>&lt;slot name=&quot;Start&quot; data=&quot;num1&quot;/&gt;</td>
<td>&lt;slot name=&quot;mrColumns&quot; data=&quot;4&quot; /&gt;</td>
</tr>
<tr>
<td>&lt;slot name=&quot;Step&quot; data=&quot;counting&quot;/&gt;</td>
<td>&lt;slot name=&quot;result&quot; data=&quot;&quot; /&gt;</td>
</tr>
<tr>
<td>&lt;/chunk&gt;</td>
<td>&lt;/goalchunk&gt;</td>
</tr>
<tr>
<td>&lt;/rule&gt;</td>
<td>&lt;/procedurechunk&gt;</td>
</tr>
</tbody>
</table>

Table 5.1: Example of a procedure and a goalchunk in XML.

The XML structure is a translation into a kind of knowledge interchange format (KIF) (cf. Genesereth & Ketchpel, 1994) agreed upon by a community of actors and enables actors to exchange and understand complex knowledge structures. The social construct, as discussed in the previous chapter, is an example of a complex knowledge structure that can be exchanged by agents to communicate norms and rules of conduct to other actors.

179
Chapter 5. RBot: Architecture and Design

Memory Component :: Social construct

In a multi-actor simulation, the interaction between actors is important as well, therefore the memory has to hold some information about social constructs. Modelling social constructs is our answer to the research question 1.2, i.e. the social construct as a memory component is a data structure for constructing social situations in the mind of the actor.

As explained in chapter 3, social constructs can be seen as representations of rules for cooperation and coordination, based on intertwined habits and mutual commitments that are often expressed in sign structures such as agreements, contracts and plans. Social constructs are reinforced by their frequent use. Hence, the chunk with its properties of activation and slots is an appropriate container for storing social constructs.

Social constructs can have some or all of the eight properties (as discussed in chapter 3): norms, (un)written, life span, authority, inheritance, scenario, context and roles.

First of all, the social construct has attached norms, which can be implemented by creating a condition-action pair: NormCondition and NormAction. Secondly, in this case the social construct is written and formalised to make it possible to implement it in the architecture.\(^17\)

The life span of the social construct in RBot depends on reinforcement of social constructs in memory; i.e. without reinforcement, the actor forgets.

Incoming chunks in perception or other memory maps are triggers for starting, reinforcing or stopping the life span of the social construct.

Authority depends on the relationship with other actors, for example awareness of other actors. When commands have to be followed up, the actor has to have an ordering of roles in combination with authority.

Inheritance and scenarios are beyond the current implementation and require more work for future releases.

Context dependency is defined by the NormConditions of the norm and they cause an automatic response to situations (context) in which the actor is involved.

The current implementation defines that social constructs are part of the actor and, being chunks, are able to be forgotten or reinforced in time. They are implemented in a way comparable to the subsumption architecture of Brooks and Flynn (1989, p. 2): “The subsumption architecture is a parallel and distributed computation formalism for connecting sensors to actuators in [actors]”. This allows the direct activation/de-activation of social constructs and norms based on changes in the perceived context. This activation/de-activation is done more or less parallel to the main cognitive processing. The activation of a norm has as effect that a number of productions get a higher or lower utility as specified in the norm action, and this is done instantaneously. This mechanism for the activation/de-activation of social constructs and the effectuation of the associated norms can be seen as a special case of a general mechanism enabling the cognitive architecture to react directly on changes in specific memory locations.

\(^17\)The unwritten social construct is based on interactions (the social level) and will be explained in the second experiment in chapter 6 that discusses the emergence of social constructs and organisation.
(e.g. the perception buffer). The mechanism consists of a listener that listens to changes in the relevant memory locations and an effectuator changing the activation, utility or other sub-symbolic parameters of productions or other chunks.

Concerning the theoretical description of the actor and the social construct theory, we only implement norms that have influence on utility of procedures. The remainder of social norms may require a more complex goal deliberating process that is not implemented yet. Therefore, in the implementation we use conditions of norms that react on changes in memorymaps of RBot, i.e. changes that can occur in parallel, together affecting the utilities in the set of procedures that are responding to the current goal. The current implementation of a social construct is shown in the following figure.

![Diagram of social construct implementation]

Figure 5.10: Classes that implement the social construct.

As figure 5.10 shows, the social construct contains a list of norms, and the Norm exists out of a NormCondition and a NormAction. The condition contains a NormListener that is connected to a memorymap. The MemoryMap notifies all NormListeners, in this case the NormConditions of all norms when a chunk is added, modified or deleted from the memorymap. The NormCondition filters the chunk and determines if it has to respond to this event and if so, it sets the Norm to active. When the norm is active, it executes the NormAction. The NormAction holds a list of procedures that are influenced by the norm. Thus, the overall effect is that the social construct creates changes in the utility of procedures and thereby promoting or demoting productions in the conflict resolution.

### 5.2.2 Controllers and handlers

The controllers and handlers section describes the control over the process and the handling of events in this process. In the kitchen example, we can compare the controllers or handlers with the cook in the kitchen who is preparing a meal. The cook first defines a goal for producing food and takes defined measurements of ingredients that are stored in the cupboards and the fridge. The cook transforms the ingredients towards meals according to recipes and either stores the end product in the fridge or serves the product to the restaurant. The cook controls the process of cooking. However many events are handled by ma-
Chapter 5. RBot: Architecture and Design

machines over which the cook has no direct control. It is the interface that delivers
the control to the cook, and thereby empowers him to indirectly handle events
that happen during the process of cooking.

The controllers and handlers of RBot have some similar functionality as the
cook. The controller has influence on the input of the recipe (procedures) and the
handlers take chunks from the memorymap and transform them and put them
back in memory, or send them out the system into messages or movements of
the actor. Hence, controllers and handlers have the function of controlling the
simulation and causing or responding to state changes in RBot.

A discrete event simulation, like RBot, is based on a time-stepping device
that controls the current time and the speed of the time steps (events). Among
time dependency, three types of handlers can be distinguished; handlers that
respond to time changes, handlers that respond to state changes in memory and
handlers that react based on messages from the server. The time dependent han-
dlers are registered with the rhythm controller and act based on time changes.
The memory state dependent handlers are registered with a memorymap, and
are notified by memory changes and the message based handler reacts on mes-
sages sent to the actor. Besides the event-based handlers, there are handlers that
take care of a specific job, for example the communication handler that deals
with the outside world and the perception handler that ranks perceived chunks
in the perception.

5.2.2.1 Goal Feeder and Goal Handler

One important aspect in a goal and procedure oriented architecture is the way
how goals are chosen, processed, do fail or succeed. Two handlers in the archi-
tecture deal with the scheduling and management of goals; the 'Goal Feeder'
and the 'Goal Handler'. The goal feeder is a decision unit that manages the
goals that are to be executed, which for example could be done based on a plan.
The current implementation is a simple FIFO buffer that handles goals based on
arrival in memory. The feeder awaits the moment the goal handler is not busy
and the goal stack is empty, so the feeder can feed the next goal to the goal han-
dler. The current implementation is simple, but allows to process a number of
goals.

The goalhandler is a finite state machine that controls the pattern matching
of conditions and firing of procedures, together with credit assignment to pro-
cedures and changes in memory. The goal handler is theoretical comparable to
the central production system of ACT-R. However some slight changes have
been implemented in the internal handling of the procedures, mainly because
the procedures have a different structure. The goal feeder and the goal handler
are shown in figure 5.11.

The figure shows that if the goal feeder contains goals, it is able to feed the
goal to the goal stack. The goal feeder is able to feed goals as long as it con-
tains goals and the goal stack is empty. In other cases, the goal feeder has to
wait for the goal stack\(^{18}\) to become empty or for delivery of new goals from the environment (e.g., other actors).

The goal handler is a kind of finite state machine that uses the goal stack as a working memory of states. The goal handler has the following cycle:

1. If there is a current goal (A) on the goal stack that has focus, then select the procedures (B) that match with the current goal. The selection is done by matching the condition of the procedure (C) that refers to the current goal.

The selection results in a set of potential procedures.

2. In case the number of potential procedures is larger than one, this is seen as a conflict\(^{19}\). The next step (D) is “conflict resolution", which results in a

\(^{18}\)The goal stack has a stack pointer that points to the current topmost data item (current goal chunk) on the stack. A push operation increments the pointer and copies the data to the stack; a pop operation copies data from the stack and then decrements the pointer.

\(^{19}\)If the set consists of one procedure, then that procedure is of course selected and there is no conflict, i.e., during conflict resolution, that procedure is automatically selected.
Chapter 5. RBot: Architecture and Design

sorted set of potential procedures (E).

3. In case the set does not contain any procedures at all (no match), then the goalhandler drops the current (sub)goal (F), a failure is credited (G) and the goalhandler returns to step 1. Otherwise, if the set is not empty, then the procedure with the highest utility is selected (I).

4. After selection of the procedure, the procedure retrieves chunks (I) from memory. The retrieval is only necessary when a retrieval is specified in the condition of the procedure.

5. When there is no retrieval error or no retrieval necessary, the engine will execute (J) the action side of the procedure, and the procedure will be credited with a direct success (K). If there is a retrieval error (L), the procedure will be credited with a direct failure (M) resulting in the removal of the procedure (N) from the set of potential candidates. After removal, the cycle is re-entered by checking if there are still potential procedures left over in step 3 of the cycle.

6. The execution of a procedure can result in a transformation of a goal (N) in the goal stack or a transformation in other memory maps (O). Three types of goal transformation (and transformation of other memory maps) can take place determined by the action statement of the procedure, i.e. modifying, adding or removing a chunk. A special case is the fact when a goal is removed from the goal stack (P); if a (sub)goal is removed and solved, the procedure(s) involved receive(s) a credit of success (K) and the solved goal is added (as a solution) to declarative memory.

7. Return to step 1.

The function of the goalhandler is to manage a number of tasks by outsourcing or delegating them to other objects in the program, i.e. sub-processes that are responsible for a certain task. Delegation of tasks to other objects increases the ease of modification, maintenance and reuse of code. Examples of tasks the goalhandler manages and delegates are:

- Goal stack.
- Pattern matching, e.g. matching a (pattern) chunk in the condition of the procedure with the goal.
- Binding of variables to values.
- Action execution.
- Management of (sub)goals.
- Credit and cost assignment of procedures.
- Administering retrieval errors.
- Managing activation changes.
The goalhandler maintains a goal stack that is responsible for ordering goals at different levels. Goals are modified, added or removed to the stack by the goalhandler, which depends on procedures that add or remove a goal from the stack (or dropping of the goal in case of a failure). The goal feeder is notified by the goalhandler when the goal stack is empty and thereby is ready to receive new goals.

**Goal stack**

The goal stack is a memorymap that acts as a LIFO buffer and keeps track of a list of goals with on top the goal that is currently in focus. It describes the division of a problem in sub problems by throwing a (sub)goal when a procedure needs to solve a goal that depends on another goal. For example, see figure 5.12, when the main goal is to go from A to B, the sub goal could be to take a step and the summation of steps creates the solution for solving the main problem.

![Goal stack diagram](image)

**Figure 5.12: Goalstack.**

**The condition side of the goalhandler**

The condition side of the goalhandler mainly deals with pattern matching of chunk(s) in the condition side of a procedure with the current goal in the goal stack. A procedure in RBot is matched in the following way. After selecting a procedure, the goalhandler selects the accompanying rule chunks of the procedure. The rule chunks consist of a mixture of rules containing references to different memorymaps (e.g. goal stack, declarative memorymap). The selected rule chunks are part of the condition as well as the action side of a procedure. The goalhandler selects the rule chunk that is part of the condition side of the procedure and has a reference to the goal stack. The next step is to retrieve the demand chunk(s) that are linked to the selected rule chunk in order to compare them with the data of the current goal in the goal stack. The first comparison is at the level of the chunktype. The chunktype gives some guarantee that the slot names in the demand chunk(s) equal the slot names in the goal chunk, i.e. the format (DTD) of the demand chunk needs to be identical to the goal chunk in the goal stack. If the chunktype of the goal chunk does not match the chunktype of the demand chunk, the procedure is not selected. Otherwise, the first test—the chunktype comparison—succeeded and a second comparison will take place. This time the contents of the slots in both chunks (demand and goal chunk) are compared. The matching of the contents of the chunk at the slot level depends on the presence of the number of slots to be matched and the presence of variables in the slots and in the slot names.
Chapter 5. RBot: Architecture and Design

There are many possibilities to match data. However, in RBot, the mathematical or logical operators that are needed for matching data are kept at the minimum, but enable RBot to do most operations necessary to resemble ACT-R. The following operators or operations are available for RBot:

- **Equal**: slot value of the chunk in the goal stack (or memory map) equals the slot value of the demand chunk; the goal condition of the procedure.
- **Not equal**: !equal, slot value of the chunk in the goal stack (or memory map) does not equal the slot value of the demand chunk; the goal condition of the procedure does not match.
- **Null**: the slot value of the chunk in the goal stack (or memory map) has to be null.
- **Empty**: it is indifferent what is in the slot value of the chunk in the goal stack or memory map. An empty slot (value) in the demand chunk (condition) of the procedure means that the procedure has no constraints concerning the value in the goal stack or memory map.

The number of mathematical expressions is kept low on purpose, because they can be handled by a combination of procedures (sub-goaling). This is in contrast to ACT-R that uses for example the function eval() and SOAR that applies expressions as for example smaller (<) than or greater (>) than. In the current architecture of RBot, we want to keep the operators at a minimum for simplicity. For specific future releases, an additional mathematical expression module (`eval` module) could easily be added to the reasoning engine.

As mentioned in the procedure component description, adding variables to the procedure gives generalisation. The goal handler builds up a list of possible variables-values combinations (binding) based on the variables specified in the procedure during the condition match. These binding variables and values are used for substitution of variables in other parts of the procedure, e.g. in retrievals or in the action side of the procedure.

Besides that, variables in RBot are used in two places: as placeholders for slot values and as placeholder for variable slot names. The syntax as described in the procedure description before is as follows:

- **=variable**: when the variable is not in the list of collected (binding) variables, the variable is assigned the slot value or the slot name value of the referenced slot of the chunk in the memory map. Otherwise, the variable is substituted with the already collected variable at the condition side.

- **!=variable**: not equal to a variable, only applied in matching when a variable is already in the list of the collected (binding) variables.

After matching all procedures with the goal chunk, three possibilities can occur:
5.2. The cognitive actor architecture

1. No procedure is suitable to execute: the goalhandler has found no suitable procedure and decides to drop (remove) the goal. If no goal chunk is left in the goal stack, the goalhandler awaits for a new arrival of a goal chunk from the goal feeder.

2. Multiple procedures are selected as possible candidate for execution: the goalhandler, because of serial processing, needs to select only one candidate. Conflict resolution helps in selecting the best procedure based on the highest utility.

3. Only one procedure is selected as a candidate. In this case no conflict resolution is necessary and the selected procedure will be executed in the next phase.

After the selection, the condition side of the selected procedure will be inspected to find out if remaining retrievals have to be made from other memorymaps. If this is the case, the goalhandler will try to retrieve the requested chunk from the referenced memorymap. If the goalhandler cannot retrieve the chunk, because of no match, or the activation of the chunk is too low to be retrieved within time, the goalhandler drops the goal and looks if any other procedure is available in the set of potential procedures. If this is the case, the goalhandler again inspects the new selected procedure to find out if any chunks need to be retrieved. This cycle will go on until all procedures fail or one procedure is selected based on a successful condition side. The next step is the execution of the procedure’s action side.

The action side of the goalhandler

The main function of the action side of the goalhandler is managing the goal stack, changing states in other memorymaps of the actor and handling the credit and cost assignment of procedures\textsuperscript{20}. In order to change memory states, the action side has three operators: remove(-), add (+) and modify(=). When applied to the goal stack, the goalhandler is able to remove a goal chunk (when a (sub) goal is solved), modify a goal chunk, or add a goal chunk in order to progress and solve a new (sub) goal. The second function of the action side is to change values in memorymaps of the actor. Chunks can be created, removed or changed, thereby changing the representation inside the memory of the actor.

Besides direct action on memorymaps, the goalhandler handles credit and cost assignment of the procedure that is executed. The procedure learns from its past experiences and thereby implements reinforcement learning at the subsymbolic level of the procedure. Next, the goalhandler determines changes in activation of chunks in memory when they are successfully harvested or when chunk merging (adding an already existing chunk) takes place, i.e. the chunk only changes in activation when it is retrieved by a production and the same production is executed as well (no retrieval errors). We will now end the discussion of the goalhandler and discuss handlers of peripheral components, such as perception, physical memory and social constructs.

\textsuperscript{20}The procedure itself is a bookkeeper of its own experiences.
5.2.2.2 Perception

The goalhandler is occupied with administering the changes inside the actor, and initiates state changes inside the memory. Changes can occur by internal cognition, but also by receiving new events from the environment. The function of the perception memorymap of the actor is to receive and store new events from the environment. The process of receiving events depends on changes of the actor itself, by physical change, or changes that are caused by the environment. The way an actor perceives objects is by receiving signs or signals from these objects, i.e. the semiotic Umwelt (Von Uexküll, 1934, 1957) as described in chapter 3.

Imagine an actor with only visual capabilities in a totally dark room. When light is added to the room, objects become clear because not only the environment is making the objects visible and colourful, but also the actor has the ability to receive the signals (photons or light packages reflected) and signs sent by the object and based on a process of semiosis, the actor interprets the sign and gives meaning to the object.

In the RBot architecture, these types of signs are represented by messages; every object is radiating signs in the environment.

However, it is only the objects or actors that are responding to them that can give meaning to the sign. At this point, the personal physical characteristics of the sensors of the actor come into play. In a dark room, a human being is unable to receive visual signs because its physical design is not capable to receive any information on the retina. However, the mosquito is an insect that has a different physical make-up of the perception. It is sensible to heat convection and smell\(^ \text{21} \), and is able to detect signals or signs that are not detectable for us. Hence our perception is only able to capture those signs that are based on the physical characteristics and make-up of the actor.

In order to create suitable perception characteristics for the actor, we have to define some physical measures of the distance within which signs can be detected (the horizon). In RBot we have defined a circle with a visual radius surrounding the actor and made the actor only sensitive to a certain segment of this circle; thus simulating a forward-oriented eye. Hence, signs can be detected in a certain area of sensitivity to signs. The second step is to determine for which signs the actor is sensitive. Because signs are represented by messages in XML, the actor is equipped with an XML parser that filters out the messages that give meaning to the actor’s personal world. The third step is to transform messages in chunk format so they can be added to the perception memorymap.

The perception memorymap is equipped with a perception handler that sorts messages depending on the predefined physical make-up of the actor. For example, in the case of RBot, signs that are received from objects closer to the actor are more important than signs further away. The handler regulates the importance of the chunks in perception with help of a mechanism that ranks

\(^{21}\)Gillett (1971) emphasizes the importance of temperature in the host-locating and homing behavior of mosquitoes. He states that warm moist air currents are the principle cues to distant host location, and that smell becomes important only in the final approach and landing stages of attack behavior. (Davis & Sokolow, 1975, p. 225)
signs or chunks. Because the perception is at the same time defined as a buffer, the signs of less importance are removed from the perception in order to reduce memory load in message overloaded environments.

In order to keep attention to crucial signs, only a certain amount of signs can get attention of the perceiving actor, which is in accordance with the bounded rationality approach (Simon, 1945). The influence of the motion and position of the actor in the ‘physical’ space on perception makes it clear that the physical properties of movement are connected to the perception of the actor.

5.2.2.3 Action :: physical memorymap and position handler

The physical memorymap has the function to store physical properties of the actor; it is the place where the actor stores its own physical properties in an ‘actor’ chunk, e.g. its name, network address, its x- and y-coordinate (2D-grid), its visual radius, and so on.

The action or execution side of procedures can steer the motor interface, i.e. they can refer to this memorymap and can change slots that hold physical properties. The simulation experiments, which will be discussed in chapter 6, use procedures that change the x- and y-coordinate in the physical memorymap. Effectors can be attached to the physical memorymap that listen to changes in the physical memorymap.

The position handler is an example of an effector that monitors the current (x,y) position in the physical memorymap and guides the actual movement when these coordinates are changed by sending messages to the server that manages the environment, see figure 5.13. The server notifies other actors that are in the vicinity or range of the moving actor, see section 5.4.2.

![Figure 5.13: Positionhandler.](image)

The change of position needs to be notified to the perception handler as well, because movement of the actor affects the perception of objects as well. The messages that are used for notifying the server are system messages, i.e. they are ‘environmental’ messages according to a format the server understands in...
order to create a change in the presentation of the position of the actor in the task environment.

Besides the system messages, there are other messages that concern the exchange of knowledge or information between actors. These messages are produced by the message handler of the actor that takes care of transforming knowledge or information (e.g. chunks) into messages that are understood by other actors that 'speak' the same language.

5.2.2.4 Action: message memorymap and message handler

In RBot, we have defined a separate memorymap that buffers outgoing messages that need to be sent to other actors. Attached to the memorymap is a message handler that reacts on so-called message chunks that are stored in the memorymap. When message chunks arrive in the memorymap, the function of the message handler is to convert chunks into messages that can be transported over a network and allows the actor to communicate with other actors, i.e. the moment a message chunk arrives in the memorymap, the message handler has the function to find out what has to be parsed into a message and orders the MessageCreator to create a message out of a formatted chunk in the memorymap.

The formatted chunk or message chunk is created by a 'message procedure', i.e. a procedure that has at its action side an action component that consists of a pattern chunk that can be filled in with properties of the receiver, the knowledge and the name of the object being sent.

For example, in the third simulation experiment in chapter 6 that is concerned with the transfer of a social construct (or goal, or procedure) towards another actor, the message chunk can have the following format/properties:

<table>
<thead>
<tr>
<th>Chunk name: message_x</th>
<th>ChunkType: Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receiver</td>
<td>actor name</td>
</tr>
<tr>
<td>Object</td>
<td>socialconstruct</td>
</tr>
<tr>
<td>Objectname</td>
<td>name</td>
</tr>
</tbody>
</table>

Table 5.2: Format of message chunk

In case of a social construct, the message handler dismantles the message chunk and asks the MessageCreator to create a message, after which the handler sends the created message away to the receiving actor(s) via the server.

The mechanism for receiving messages happens in a similar but reversed way. An actor sends a message to the receiving actor via the server. The message arrives in the message memorymap or buffer of the receiving actor in a language or format that is understood by the receiving actor, i.e. the actor receives messages in its message memorymap or buffer and parses the message/transforms the message into internal chunks or commands understood by the actor's architecture. The communication and formatting of these messages will be discussed in detail in section 5.4.1.4.
5.2. The cognitive actor architecture

5.2.2.5 Memory components and estimating the sub-symbolic properties

Up to this point, the structure of the memory has been explained and the necessary handlers for regulating state changes and receiving input from the perception and sending messages to other actors or the server have been discussed.

However, some issues concerning administering and estimating or calculating the sub-symbolic properties need to be addressed as well. Two software classes, the ActivationMath class and the SubSymbolicMath class are responsible for calculating the activation of a memory component and the sub-symbolic properties of procedures at a certain time.

A memory component keeps track of the activation of a chunk and maintains an array of presentation times of itself. The length of the array can be configured during start up for every actor and depends on the amount of working memory available in the computer and the configuration of the decay factor\(^{22}\). An example array with a length of four places is given:

\[3, 7, 12, \emptyset]\n
When a handler, such as the goalhandler wants to know the activation of a memory component, the memory component calls the ActivationMath class that calculates the activation of the component based on the base-level learning equation for activation. When the current time for example equals twenty discrete time steps, the activation of the memory component is:

\[\text{Activation} = \ln \left( (20 - 3)^{-0.5} + (20 - 7)^{-0.5} + (20 - 12)^{-0.5} \right) = 0.87\]

Associative strengths of ACT-R are also implemented in the ActivationMath class. One of the three methods of calculating activation spread\(^{23}\) can be applied in RBot at the moment the goalhandler wants to retrieve a chunk from memory.

The other class, the SubSymbolicMath class, takes care of the sub-symbolic properties of the procedure chunk, computes the requested property, for example the expected gain of a procedure, and returns the calculated property back to the component that requested the computation of a property.

Both these classes that handle sub-symbolic properties have an indirect influence on the performance of the cognitive engine. The social construct handler, discussed in the next section, also has indirect influence on the performance of the cognitive engine, but its mechanism (a meta-cognitive mechanisms) is different compared to the sub-symbolic cognitive mechanisms.

5.2.2.6 Social construct handler

All previous handlers are directly dependent on notification of time. Social constructs on the other hand do not only depend on the time aspect, but also respond to changes in the situation inside the memorymap(s) for which it has registered itself. A social construct can register itself for example at the declarative

\(^{22}\)The larger the decay, the smaller the chosen length of the array. Because, with growing time difference between current time and a presentation, and a large decay, the influence of the presentation gets smaller and can eventually be neglected.

\(^{23}\)See section 4.4.3.1 in chapter 4.
memory as a listener for events. When something happens inside the declarative memory by which the social construct is triggered, the social construct handler determines if the event needs to result into an action, e.g., in the form of a change in the utility of procedures that are known to be affected by this construct.

Norms are attached properties of the social construct, and they can be explicitly formalised. The condition side of the norm can be triggered depending on:

- A chunktype or slot condition of a chunk to which the condition should respond
- The registered memorymap, a map to which the condition side is listening, it only listens to events created in this memorymap.
- A trigger time: the condition side is only triggered when the condition is still present after a certain waiting time.

When the condition side is triggered, the action side is fired. The action side can have impact on many components in the model. For example, the action could affect goal deliberation, other norms, affect the utility of procedures, or generate other actions such as sending messages to certain actors. We refer to chapter 3 that discusses a number of social constructs with attached norms, and their possible effects.

In the current implementation, social constructs are able to change the utility of procedures directly without reasoning so that they have an immediate effect on behaviour, see figure 5.14.

Figure 5.14: Social constructs and the impact on utility of procedures.

For instance, in figure 5.14, we see the influence of the social construct on procedures P1 and P2. When a chunk is arriving in a memorymap, the norm condition is notified and checks whether it can trigger an action. If the condition is true, the norm becomes active and takes action by changing the preferences of P1 and P2. P1 is, for example, demoted and P2 is promoted.
5.2. The cognitive actor architecture

The section about controllers and handlers has shown how the architecture handles events and how this can be controlled. The next section explains the view of the actor RBot that enables the user to have control over the actor.

5.2.3 View and control of the model

The view of the actor RBot is implemented in a user interface (figure 5.15). We will discuss the most important functions that allows the user to (indirectly) control the stand-alone RBot actor24.

![User interface of RBot: view and control.](image)

In RBot, as in most simulation environments, the user interface allows the user to control the simulation parameters as follows:

1. **Before running the simulation**: configuring the actor and simulation parameters. Initializing a model can be done by configuring the actor’s properties with the help of configuration files; files formatted in XML that can be edited with an XML editor.

   Another important aspect is the configuration of the RBot simulation parameters (based on ACT-R) before launching an actor. Examples of such parameters are the decay rate of the memory components, the noise of the utility function, and so on.

   After launching the actor, the XML configuration files can be selected and loaded into the memory of the actor, i.e. the procedures file, declarative facts file and the goals for the goal feeder are loaded into the memory of the actor by parsing the files and transforming the files into chunks. After configuring the actor, the simulation can be run.

2. **During the simulation (control and inspection)**: the simulation can be controlled by starting, pausing, and stopping it and making a simulation step

---

24The user-interface of the MRS with more actors will be discussed later on in section 5.4; the mixed tier.
Chapter 5. RBot: Architecture and Design

at a time. During the simulation, it is possible to inspect various parts of the actor. For instance, the Memory View, as shown in figure 5.15, allows the user to inspect the contents of the memory—chunks, procedures, social constructs. Another option is to look at the procedure-trace that allows for inspection of the interaction between the goal stack and productions. The debug option allows checking the handlers (e.g. the message handler) with help of traces (messages sent and received) in a debug panel.

3. Presentation of output (graphs and files): the output of the information is stored in files and in database tables. After the simulation is finished, files are generated\footnote{This is also a configuration parameter, which can be set to choose which output files need to be generated.} that can be read with a text-editor. An example of such a file is the file generated by the goalhandler that shows the trace of the goalhandler cycle.

The data in the database tables represents values, such as utilities and activations of memory components. The presentation of data is handled by a tool whose user interface is integrated as part of the user interface of the mixed tier, see section 5.4.3.

The main purpose of the user interface is to configure RBot in a user friendly and convenient way, inspect the actor's behaviour and get results after the simulation finishes; results that can be used for explaining the behaviour of actors. Some of the output will be stored in files and other output in a database or backend server from which data can be extracted for presentation in tables or graphs.

5.3 Backend server

During every simulation run, data can be stored in files or in a database. This is necessary when simulation data needs to be analysed after the experiment finished. Every actor has a direct connection to the database (see figure 5.16, model 1) to prevent additional overhead in the network; in opposite to the alternative, in which data is first sent to the server and then stored in the database (see figure 5.16, model 2).

Moreover, model 1 allows the actor to run stand-alone without the server and store its data in the database. Secondly, the actor is independent of the server and allows it to function as an actor in a Point-to-Point environment (see section 5.1.2), i.e. it can possibly act as a client as well as a server in a multi-actor environment.

The user has the possibility to select what data of the actor needs to be stored, e.g. activation of memory elements or position and time data. When the simulation starts at time zero, the actor first stores its begin configuration in the database. Subsequently, the actor stores every time step data, only when changes in the memory or other events occur; this in order to keep the amount of data being transported at a minimum.
In the current implementation, the actors have the function to store and not present their stored data. The mixed-tier is responsible for reading the data from the database and presenting tables or graphs to the user based on the data.

5.4 The Mixed-Tier

The mixed-tier component, i.e. the server, the task environment and the data presentation, has been designed as a central component. The mixed tier is responsible for control and visualisation of the task environment, and presentation of the output of the (inter) individual experiences of the actors and the environment in which they live. Moreover, the advantage of central control is that the complete simulation can be controlled from one control centre instead of controlling every actor separately; even when all actors are dispersed over multiple computers. The mixed-tier fulfils several roles, i.e. it has three main tasks:

Server: control, registration, synchronisation and communication The first task is to control the properties of the environment, registering new actors, and thereby allowing the server to control the synchronisation of processes between the distributed actors. The communication is controlled by the server and regulates (1) the (environmental or system) communication and synchronisation between actors and the task environment, and (2) the communication (ACL\textsuperscript{26}) between actors.

Task Environment The mixed-tier also takes care of representing the task environment and visualising and communicating the environment to both, the actors and the users that see the environment being presented on the computer screen.

Graphs and data presentation The third task of the mixed-tier is to act as a client of the backend server. The client creates queries to be executed by the database that returns the correct data for presenting graphs. The data and graphs allow the researcher to interpret the behaviour of individuals and the changes of memory structures of each individual actor.

\textsuperscript{26} Actor (or Agent) Communication Language
5.4.1 Server: control, registration, synchronisation and communication

As mentioned before, the server controls and determines the properties of the environment, takes care of the registration, synchronisation of processes and any communication that takes place between actors or actors and task environment. The purpose of central control is to get complete control over all the events that eventually may happen during a simulation run.

5.4.1.1 Control of the simulation task environment

The control of the server is restricted to the environment in which the actors live. The (cognitive) processes internal to the actor are in control of the actor itself, thereby giving the actor as much autonomy as possible.

Most of the control is possible during the initialisation phase of the simulation. Before the simulation starts, the user is allowed to specify the properties of the task environment, e.g. determining the number of actors that can take part in a simulation run. After registration of the actors, the only control the server has is the synchronisation of time and processes between the server, the actors and the task environment. Apart from that, it is possible to start, pause or end the simulation. But before any actor can be controlled, it first has to register itself.

5.4.1.2 Registration of actors

When the simulation starts, the server will first start up the task environment with its environmental parameters, start up the view and controller, and will wait until an actor connects to the server. After starting the server and task environment, an actor has to register itself in order to be represented in the task environment. The registration process is shown in figure 5.17, in which there is first a connection at the level of the network and after the established connection, the actor sends a ‘register’ request to the server.

The server parses the incoming message and decodes it as a register request. The request contains information regarding the physical properties of the actor that enables the server to create a symbol of the actor. The symbol is added to the symbol attribute map, after which the server is able to create a drawing of the actor in the task environment. The server sends a confirmation back to the actor, causing the actor to be aware of being registered at the server and being able to receive messages from the server concerning the physical environment and sending or receiving messages from other actors.

5.4.1.3 Synchronisation of processes

To control the behaviour of actors that can be distributed over many computers, and in order to replicate experiments in a controlled way, it is important that all actors become aware of the same time and the notion of environmental events and other actors’ events.
Synchronising processes and events can be established with help of a simulation clock. If each independent process publishes its current advance in time with help of a simulation clock, then it should be possible to synchronise these parallel processes. In order to decide how to synchronise processes, we first discuss the simulation clock.

There are two principal approaches for advancing the simulation clock: next-event time advance (or discrete event simulation) and fixed-increment time advance (or discrete time simulation) (Law & Kelton, 2000; Mitrani, 1982). The next-event time advance approach is shown in figure 5.18.

With the next-event time-advance approach, the simulation clock is initialised to zero and the times of occurrence of future events are determined. The simulation clock is then advanced to the time of occurrence of the most imminent (first) of these future events, at which point the state of the system is updated to account for the fact that an event has occurred, and our knowledge of the times of occurrence of future events is also updated. Then the simulation clock is advanced to the time of the (new) most imminent event, the state of the system is updated, and future event times are determined, etc. (Law & Kelton, 2000, p. 8)
Hence, in figure 5.18, the events \( e_1, e_2, e_3 \) determine the simulation clock advance \( (Sim_1, Sim_2, Sim_3) \), i.e. a discrete event simulation makes use of such a simulation clock. On the other hand, with the fixed-increment time advance, see figure 5.19, the simulation clock is not determined by events, i.e.

...the simulation clock is advanced in increments of exactly \( \Delta t \) time units for some appropriate choice of \( \Delta t \). After each update of the clock, a check is made to determine if any events should have occurred during the previous interval of length \( \Delta t \). If one or more events were scheduled to have occurred during this interval, these events are considered to occur at the end of the interval and the system state... [is] updated accordingly. (Law & Kelton, 2000, p. 9)

\[ Sim_0 = 0 \quad Sim_1 = \Delta T \quad Sim_2 = 2\Delta T \quad Sim_3 = 3\Delta T \]

![Figure 5.19: The fixed-increment time advance.](image)

With fixed-increment time advance, the simulation clock is independent of the events \( e_1, e_2, e_3 \) that happen during the simulation, i.e. the events are accumulated and calculated at the end of the period.

From both time-advance mechanisms, the next-event time advance is the most applied, because the fixed-increment only is useful for simulating events that are scheduled to be released regularly (controlled), e.g. the expiration date of options in economic simulations every month.

In a Multi-Agent System, we are dealing with a number of different simulators (actors) that need to be synchronised with the task environment and each other. Because actors may run at different computers and may do different tasks, they may easily get out of sync with other actors and the task environment. In case of simulation of multiple processes, we take a next-event time-advance approach in which the synchronisation of the clocks of all registered actors is an event that determines the simulated time, i.e. the simulation clock is a multi-actor synchronisation clock (see figure 5.20).

Figure 5.20 shows two actors, both with a next-event time line. In the first period \( (\Delta T) \), actor 1 and actor 2 both generate three (not necessarily similar) events \( (e_1, e_2, e_3) \). Actor 1 consumes more time than Actor 2 and synchronisation takes place after Actor 1 has finished its events. In the second period \( (\Delta T') \), we see that Actor 2 generates an additional event and needs more time than in the previous period. The simulation time in period \( \Delta T' \) is determined by actor 2, after which again synchronisation of events takes place. During the simulation, the actors are running in sync, insofar that internal events (e.g. the cognitive
engine) are not controlled by the synchronisation clock and are independently executed.

The purpose of the simulation clock is to keep the actors as much as possible in synchrony. Otherwise, one actor could experience a state change (of the environment) a fraction of a time earlier before the other actor would. An actor out of sync with the environment or other actors could perceive the world as it would be a while ago or as it would be a minute from now. Such difference can have influence on what is being perceived by several actors at the same time, i.e. each actor could perceive a different world surrounding him and react in a different way as normally would happen in a synchronised world. Therefore, synchronisation is important, and to keep processes of actors as much as possible synchronised, we have implemented a timer communication mechanism that distributes time to actors based on a time-protocol, see figure 5.21.

![Multi-actor synchronisation clock](image)

**Figure 5.20: Multi-actor synchronisation clock.**

When actors register themselves at the server, they register at a pool of actors that is monitored by the server. The server tries to keep the actors synchronised...
by keeping the times of the actors in this pool synchronised with help of a time protocol. The protocol is a double commit protocol in which the server asks the actors to prepare themselves for a time change. When all actors are ready, a timer update message is sent to the actors and when the actors have sent the confirmation, the process starts over again. Controlling and synchronising the time of actors is one of the difficult aspects in truly independent actors, caused by, amongst others, the following physical conditions:

1. Actors can live on different machines with different processor speeds that can create differences in execution and response times.
2. Small delays in network and serial queueing can easily cause actors running out of sync.

Syncing actors and keeping the awareness of time equal between many actors is a study beyond the aim of this dissertation, but is something that certainly needs to be addressed in studies about distributed real-time systems.

As explained before, in the design of RBot, the actors are running closely in sync with help of a simple double commit time protocol. The communication that is necessary to let the time-protocol function is described in the next section.

5.4.1.4 Communication among actors & between actors and task environment

The main job of the server is to create a task environment—semiotic Umwelt—in which actors can communicate and perceive each other. This allows them to exchange social norms and other events with the help of perceived actions (gestures) and the exchange of an actor language.

As mentioned before, the communication between actors and task environment can be divided into two types of communication.

1. Communication that controls the synchronisation process between actors and the environment; and
2. Communication between actors in a language that is understood by all actors participating in the task environment.

Communication between actors and task environment

For actors to become aware of time and space, they have to interact with the task environment to handle changes that occur in the task environment. The first, as we already have stated, is the awareness of time that helps synchronising the actor with the task environment. The other issue is the physical awareness of space and objects in the task environment. This awareness is controlled by the server that sends (environmental or system) messages towards the actor when another object comes in the visual range of an actor.

Objects in the task environment are changing due to actions of the actors in the environment—e.g. the actor sends messages the moment its location changes—or the environment itself is changing, e.g. weather change. The communication—controlled by the server—between the actors and the task environment will be elaborated in section 5.4.2.
Communication among actors: an actor communication language

In chapter 3, we have analysed the message-channel as a possible medium for the transmission of someones intentions or knowledge to other actors. We have stated that communication should not only exist out of transmission alone, but also should include the social context and a signification system.

In our MAS, we implement the transmission of signs with help of a standard language (FIPA-XML) that exists out of speech acts. As explained in chapter 3, messages can have the following form, see table 5.3.

| (performative
  : sender actor x
  : receiver actor y
  : content (...)
  : language XML
  : ontology Social Construct DTD) |

Table 5.3: General markup of the actor message.

The performative (e.g. inform or request)\textsuperscript{27} defines the communicative or speech act that states what the sender's intentions are (and what to expect from the other). As explained in the section about the message memorymap, the messages can for example consist of social constructs, goals or procedures. An example of the content of an RBot message that holds a social construct is shown in table 5.4.

| <socialconstruct name="rightlanedriving">
  <norm name="rightsidelane_norm">
    <normcondition name="rightsidelane_condition" type="Actor">
      <memorymap name="perception" triggertime="0"/>
    </normcondition>
    <normaction name="motivate_evasive_right">
      <affect name="evasive_maneuvreRight"
        target="procedure" motivation="10"/>
    </normaction>
  </norm>
</socialconstruct> |

Table 5.4: Content of RBot message.

The language used for expressing the message is XML and the DTD associated with the message is the 'social construct DTD'. The receiving actor has a signification system that parses the content of the message (e.g. the content shown in table 5.4) based on the social construct DTD and creates a social construct in its memorymap that has the following meaning for the receiving actor.

\footnote{\textsuperscript{27}See http://www.fipa.org/specs/fipa00037 for a complete list of FIPA performatives.}

201
Chapter 5. RBot: Architecture and Design

When a chunk of type ‘Actor’ is perceived in the perception map, then (triggertime = 0) the norm action is immediately triggered, which states that the target procedure with name ‘evasive manoeuvreRight’ should be stimulated with a motivation of 10.

Besides the transfer of signs and the signification system, the task environment (medium) also has impact on the communication between two actors. The communication between actors can be closely linked with the properties of the environment. For example, when an actor wants to say something to another actor, it should produce high enough volume (a physical property of the sending actor) to reach the other actor. Still, whether the other actor can receive the message, depends on the radius in which he still can hear (also a physical property).

In our current version of RBot, the model of speech is defined very simple and a message can always be received by the other actor as soon the actor sends the message towards the other actor. The task environment will be discussed in the next section that elaborates more on how actors perceive each other, which is also based on messaging.

5.4.2 The task environment

The current task environment, see figure 5.22, is created specifically for the experiments described in the next chapter. Hence, a task environment should be redefined for every new and specific simulated environment. For example, the well known world of blocks (Minsky, 1985, p. 21) should be modelled and graphically designed for that specific environment.

Direct feedback (especially 2D and 3D graphics) about the progress of interaction between actors gives better options for debugging the simulation. Moreover, as observer, it is easier to estimate whether the actors behave according to the design.

The construction of the view works the way a monitor builds up its screen. The environment (or game) panel has a refresh rate and during one refresh the environment is constructed, e.g. the grid, the actors and there properties and the connections between actors. The view will not change as long as the underlying data is not changed.

In the task environment, perceiving others is defined by senses (a semiotic Umwelt). The human being has the following senses: touch, smell, taste, hearing and vision. Vision is the most important perception organ for perceiving the environment (especially when a computer gives visual feedback), therefore we have concentrated our efforts on vision in the current version of RBot. Vision is implemented by creating a visual radius around the actor that determines what objects fall within the perception range of the actor. The perception range is further delimited by the actor’s eye angle that is focused towards the direction the actor faces.

---

28 Such an explicit graphical design is not always necessary and could be simulated with help of matrices of data points.
5.4. The Mixed-Tier

![Figure 5.22: A task environment in action.](image)

The stimuli that are sent to the actor are determined by the environment and not by the actor itself. The server determines whether an object or actor is within the perception range of the actor and sends a stimulus in the form of a chunk to the actor that holds information about this perceptible object or actor. The actor filters this information with help of the perception handler in order to react only to messages that are of importance to the actor.

In the case of hearing, the actor has a single centred ear that receives signs that fall within the aural radius. The speech radius determines how far communication can reach, i.e. the “sound” the actor produces has a certain range in which other actors are able to hear it. Hence, the actor has to be in range of the other actor’s speech radius to hear what it says. In our architecture, the environment and the distribution of signs (messages) are controlled by a central server that manages the spreading of signs to other objects in the environment, see figure 5.23.

![Figure 5.23: Server controls who perceives what.](image)
Chapter 5. RBot: Architecture and Design

The environment is constructed with help of messages that carry information necessary to draw actors and objects in the environment. The actor sends messages that concern its initial physical properties, the moment it registers itself at the server. After registration, the actor only sends messages (arrows directed towards the server in figure 5.23) when its physical properties change, e.g. the actor moves from position \((x_0,y_0)\) to \((x_1,y_1)\).

The server on its part only sends messages (the striped line with arrow pointing to actor A in figure 5.23) when they are relevant to the actors or objects.

In case of figure 5.23, the server knows the locations of the actors and therefore is able to calculate the distance between actor A and actor B. Assuming the server has received physical properties from both actor A and actor B, it is able to determine that only actor A requires a message indicating the presence of actor B; actor B is not made aware of actor A. After actor A receives the message of the server, it is filtered by its perception handler that concludes that actor B is perceivable, otherwise it ignores the message.

Another aspect the server takes care of is the process of transforming data describing the physical properties (e.g. position, size...) of the actors in the environment into graphics that can be displayed on the screen.

A relatively simple case is when an actor sends information about its physical properties towards the server. The server sends the data towards the task environment that immediately draws the image to the screen.

A more complex case is the previous one projected in figure 5.23. The following figure (figure 5.24) shows what happens during the moment an actor perceives another actor and the moment the actor moves out of sight of the actor.

![Diagram](image)

Figure 5.24: Communication between actor and environment.

Every time step, the server requests the environment to update the drawing (sprite) of every actor. The data (physical properties) on which the drawings are based, are stored in the memory of the environment as symbols or attributes.
Every time step, every drawing object wonders if there are other symbols (drawings) that overlap with themselves.

Three situations can happen: (1) the moment the drawing object discovers that another object falls within range and the symbol of that object was not in range before (not already connected), then the symbol is added to the set of symbols that hold all other symbols (drawings) that are in range, (2) the drawing was already part of the set and only the symbol properties have to be modified, and (3) the other object moves out of the range and therefore the associated symbol is removed from the set of connected symbols (drawings).

The previous estimation whether an actor is in range of another actor is handled by the environment. When such an event happens, the environment sends an XML formatted message with the physical properties (e.g. the position) of the other actor (or object) via the server to the relevant actor. The actor that receives the message is equipped with an environmental message parser that transforms the message to an addition, modification or removal of a perception chunk in the perception memorymap.

This scenario describes what the function of the environment is: (1) keeping track of physical properties of objects in the environment, (2) estimating whether actors or objects in the environment collide in the environment, and (3) create drawings of every object in the environment. The number of events that can happen in an environment are many more, such as changes that are independent of actors, e.g. weather, times, differences in height.

However, the modelling of a complex environment is beyond the aim of the dissertation. The creation of the task environment only has the purpose for supplying a simple arena that allows simple experiments as described in chapter 6 to be displayed.

The next section will cover the client that is part of the mixed-tier interface, which is concerned with graphs and data presentation.

### 5.4.3 The client: graphs and data presentation

The last function of the mixed-tier we want to discuss is the data presentation. Based on the results that are extracted from the database in the backend server, the researcher can look back in history and estimate what actually happened during the simulation. The interpretation of data is important, and can be simplified by developing a client with a user interface that queries data from the database and presents the results in graphs or tables to the user.

The structure of the data in the database allows the storage of a set of experiments. Each experiment is assigned a unique experiment identifier. Within the experiment, each actor gets assigned a unique actor id. These identifiers allow the creation of queries that retrieve data from a specific experiment and a specific actor. The following figure (figure 5.25) shows the history of a procedure that was three times successfully applied during its life time.

The current implementation of RBot (part of MRS) allows for storing and presenting data of different levels of the actor; intra-individual data—the sub-symbolic data as shown in figure 5.25, individual data of the actor—e.g. its physical properties and its memory structure, and inter-individual data—e.g. data of
interaction between actors.

A multi-actor simulation creates many events that can be stored in a database or in files. Therefore, in order to prevent overloading a computer system, choices have to be made about what needs to be measured before asking for too much data that unnecessary slows down the simulation.

In the next chapter, we will show results in the form of graphs that are created with the help of data that has been collected during a set of experiments. The user interface was the last component of the design that needed to be discussed. In the next section, we will discuss the current design and see if it does fulfill the design requirements we have stated in the beginning of this chapter.

5.5 Discussion

In the discussion we want to reflect and see if our design meets up to the design requirements. Any suggestions to extend or change the architecture, will be postponed to the discussion section or further research in the conclusive chapter 7.

The first requirement states that the architecture should be a Multi-Agent System whose behaviour should be explained in terms of individual autonomous components (actors) and their interaction or relations between each other. Our design fulfils the autonomous aspect by creating individual actors that have their own cognitive engine that makes autonomous decisions based on representations in memory and individual experiences. In a pure classical cognitive architecture, these decisions are made completely independent from other actors. In our architecture, the client server pattern (by creating a semiotic Umwelt as a separate entity) allows for interaction between multiple actors and is an appropriate design for an architecture that can implement interaction mechanisms and delivers a basis for creating social situatedness.

In our opinion, the first requirement has been fulfilled with our design. In
the next chapter, the second experiment will show that actors interact with each other and make autonomous decisions based on their experience with the environment.

The second design requirement demands that a social construct should be a representation in the mind of the actor that can guide social behaviour. In other words, the social construct is a unit of knowledge that allows an actor to respond to certain physical and social situations. A social construct regulates behaviour of the actor at a meta-cognitive or normative level, i.e. the social construct listens to changes in the memory of the actor and gets triggered, the moment a situation appears for which the social construct is applicable. In our design, we have integrated aspects of Brook’s subsumption architecture (Brooks, 1986), an architecture that is concerned with the physical environment of the actor and enables the actor to respond with behaviour patterns that match certain situations.

We have extended this idea by not only including physical, but also social situations. The data structure consists of a condition—that listens to changes in the physical or social situation—and an action. The condition-action pair is at the level of meta-cognition, and acts in parallel with the cognitive engine at the lower level. The integration of the subsumption idea allows for an implementation of situated cognition based on ACT-R. Such a modular structure allows for easily adding, modifying, removing and exchanging social constructs with other actors.

The next chapter shows an experiment in which an actor transfers a social construct to another actor. This experiment demonstrates that the subsumption idea applied to an actor is a good choice for implementing coordination mechanisms, such as social norms or contracts between actors.

The last requirement concerns the implementation of a cognitive plausible actor. According to Newell (1980), a physical symbol system is required to hold representations. The ACT-R architecture (Anderson & Lebière, 1998) is such a physical symbol system and is a (empirically tested) cognitive plausible architecture that fulfills such a requirement.

In our design, we have adopted most of the cognitive mechanisms and structure of ACT-R, but enhanced the design of the memory structure and its components. The addition of meta-cognition and an ACL language parser allows the actor to communicate with other actors and be physical and social situated in the environment, i.e. the actor responds better to influences from the outside world than ACT-R can.

The first experiment of the next chapter shows one of the many possible experiments for comparing ACT-R with RBot. The experiment will not show a detailed comparison, but will show that the behaviour of both architectures is similar.

Although cognitive psychology requires architectures like ACT-R that are precise in simulating the behaviour of the individual, other fields such as sociology and social psychology require architectures that show how individual behaviour is affected by the environment and behaviours of other individuals. In our opinion, RBot (and MRS) allows for studying behaviour of not only individuals (cognitive psychology) but also social behaviour and interactions between actors.
Finally, we want to address the innovations we have applied to ACT-R that has resulted in RBot and MRS.

1. The first innovation is the access and structure of the memory. The implementation of the memorymap, chunks and links in RBot allows for much more complex and flexible structures than ACT-R.

2. Procedures in RBot are small networks of chunks, i.e. chunks are the basic elements in RBot. RBot is therefore more unified than ACT-R that contains separate production structures.

3. The third innovation is the implementation of the variable slot name (DSN chunk). Although ACT-R is discussing the issue, an implementation has not been established yet (version 5.0).

4. The introduction of social constructs in RBot as units of knowledge that are applicable to different situations (physical as well as social) and the integration of aspects of the subsumption architecture is a complete new feature. Discussions about for example meta-cognition are present in the ACT-R community. However, the implementation is not there yet.

5. The formatting in XML is new compared to ACT-R (SOAR already has XML formatting). The formatting of memory structures that allows the memory (components) to be stored and transported over a network is not implemented in ACT-R.

6. Another aspect we want to mention is the creation of a MAS (Multi-RBot System) that consists of a task environment in which more than one actor can live and interact with other actors. Interaction creates new possibilities of not only studying the behaviour of the individual alone, but also the interaction between individuals at the social or organisational level.

7. The final aspect is the introduction of the semiotic Umwelt. The actors are able to perceive the environment and each other by exchanging signs with each other and the environment, i.e. the physical changes are exchanged (communicated) with the environment as well as the exchange of messages (e.g. social constructs, goals and so on) between actors. The semiotic Umwelt creates a (sign-based) environment for actors, and thereby supports physical and social situatedness.

The current chapter explained the design of RBot and MRS with which we tried to fulfil the requirements we have stated in the beginning of the chapter. Right now, the design only has theoretically proven itself. However, such requirements need to be tested and validated with help of demonstrative experiments. This will be done in the next chapter.