Chapter 5
Multi-agent Systems

Juan C. Burguillo

The term agent derives from the latin participle *agens*, coming from the verb *agere* that means *to do* and denotes the capability of an entity to do or to act. But in practice the term refers to multiple meanings in different contexts, like a human agent, a hardware agent, a chemical agent, etc. In this book we are interested in the concept of autonomous agents, and for that purpose one of the classical and simple definitions comes from [21] and states that: *an agent is any entity that can perceive its environment through sensors and change it by means of actuators*. Under this agent conceptual image we still could find many simple hardware or software objects that, being rather deterministic, would fit within the agent philosophy. A more general discussion and description about autonomous agents appears in [28]. According to it, the term agent refers to a hardware or (more usually) software-based computer system that usually enjoys the following properties:

- *Autonomy*: agents operate without the direct intervention of humans, and have some kind of control over their actions and internal state.
- *Social ability*: agents interact with other agents (and possibly humans) via some kind of agent-communication language.
- *Reactivity*: agents perceive their environment, and respond in a timely fashion to its changes.
- *Pro-activeness*: agents do not simply act in response to their environment, they are able to exhibit goal-directed behavior by taking the initiative.

There are some other attributes that can be present, but usually they are not considered as a requirement: mobility, veracity, benevolence, rationality and adaptability (or learning) (see [28] for a more detailed discussion).

Agents coexist and interact with other agents in different ways. A system consisting of an interacting group of agents is called a Multi-agent System (MAS). Within

J. C. Burguillo (✉)
Department of Telematics Engineering, School of Telecommunications Engineering, University of Vigo, 36310 Vigo, Spain
e-mail: J.C.Burguillo@uvigo.es

J. C. Burguillo, *Self-organizing Coalitions for Managing Complexity*, Emergence, Complexity and Computation 29, https://doi.org/10.1007/978-3-319-69898-4_5

the context of this book, a complex software system can be treated as a collection of many agents, each one with its own local functionality and properties. Some of the benefits of MAS technology in large scale software systems are [24]:

- Speedup and efficiency, due to asynchronous and parallel computation.
- Robustness and reliability, in the sense that the whole system can undergo a *graceful degradation* when one or more agents fail.
- Scalability and flexibility, since it is easy to add new agents to the system.
- Cost, assuming that an agent is cheap compared to the whole system.
- Development and reusability, since it is easier to develop and maintain modular than monolithic software.

So far, multi-agent systems have been applied in different domains. Some examples are:

- Social sciences, where MAS technology is used for simulating interactivity and other social phenomena.
- Distributed data mining and information retrieval.
- Virtual reality and computer games use agents to implement intelligent behavior.
- In robotics, a common application is to manage a group of robots, so that they can locate themselves to navigate through their environment.
- Internet auctions and e-commerce.
- Management of telecommunication networks.
- Artificial life and biological simulations.

These are just some examples, since MAS have also been applied to control, scheduling and planning of manufacturing, air traffic management, medicine, e-Learning, etc. The interested reader can find examples of MAS applications in these domains in [27].

5.1 Short Historical Notes

Historically, the origins of autonomous agents and multi-agent systems trace back to the 70s with the works done by Victor Lesser [7, 15] that state the cooperation of multiple entities in order to solve complex problems. Such cooperation help to jointly solve tasks that could not be solved by individual entities, or at least to provide a higher solution efficiency. These works somehow complement the classical centralized approach followed by many AI researchers on those days, more centered on expert systems or knowledge-based systems that derived in Knowledge Engineering. Some of the problems faced on those days were the limited context of application, the difficulties to model the context to reason with intelligence, and the difficulties to exchange knowledge with similar entities, i.e., to cooperate. Then, distributed approaches appear to enhance and extend the capabilities of classical isolated and centralized systems. The design and development of such cooperating systems face

several challenges, like how to model knowledge to be exchanged among several entities, how to coordinate effectively the distributed entities, and how to obtain a higher efficiency than centralized systems.

Such working area within the AI community was called Distributed Artificial Intelligence (DAI) and considered several approaches to model, and to solve complex problems in AI involving reasoning, planning, learning, etc. The conditions for applying DAI techniques show characteristics like large data samples, distributed systems and/or loosely coupled autonomous processing. Distributed Problem Solving (DPS) and Multi-agent systems (MAS) are the two main approaches resulting from DAI.

- On the one hand, distributed problem solving techniques divide the tasks among a set of nodes, and the knowledge is shared. Its main concerns are task decomposition, knowledge synthesis and how to provide a global effective solution.
- On the other hand, in multi-agent systems, agents provide a more flexible and autonomous approach to coordinate their knowledge and activities, in order to model the problem to be solved, and this may involve cooperation or competition among different groups of agents.

Multi-agent systems become an excellent approach to model and simulate complex systems as they can apply the traditional top-down approach of AI as well as a bottom-up approach, becoming a good vehicle for the emergence of new system properties, which are one of the basic characteristics of complex systems. Along the next sections we introduce in a more detailed perspective the main characteristics of multi-agent systems.

5.2 Intelligent and Autonomous Agents

An autonomous agent is usually modeled by a certain system architecture. In this section we first introduce the three basic agent architectures, and from them we consider concepts related with the architecture of the whole multi-agent system.

5.2.1 Deliberative Agents

Deliberative architectures use a symbolic knowledge representation, where agents start from an initial state, and then generate plans in order to achieve their goals. Therefore a deliberative agent has a symbolic model of its environment, and decisions are taken considering logic reasoning mechanisms and symbolic manipulation.

The main difficulty when using deliberative architectures is to provide and adequate symbolic representation of the problem, because; given the complexity of symbolic manipulation algorithms, it is possible that the agent cannot face the real time requirements inherent to most of real world environments.

The BDI (Beliefs, Desires and Intentions) architecture [20] has been the most popular deliberative architecture in the past years, and there are a relevant number of implementations. This agent architecture considers a philosophic model based on Michael Bratman's theory of human practical reasoning. The basic concepts related with the BDI architecture are:

1. *Beliefs*: represent the knowledge that the agent has about its environment. This knowledge can be modeled as variables in a database, symbolic expressions from predicate calculus, or inference rules to create new beliefs.
2. *Desires*: represent what the agent wants to accomplish, i.e., some of them can become the next goals or the final objectives that the agent will pursuit.
3. *Intentions*: describe what the agent has chosen to do, and they usually include a set of plans oriented to achieve the agent desires. These intentions should be changed appropriately in dynamic environments if the agent perceive that the context has evolved and one or more plans are not valid anymore.

Besides these three main BDI elements, the system usually works based on *events* that may update beliefs, trigger plans or modify the agent goals, p.e., when some of the desires becomes unfeasible. Algorithm 1 describes the typical cycle for the BDI interpreter as described in [20]: "At the beginning of every cycle, the option generator reads the event queue and returns a list of options. Next, the deliberator selects a subset of options to be adopted and adds these to the intention structure. If there is an intention to perform an atomic action at this point in time the agent then executes it. Any external events that have occurred during the interpreter cycle are then added to the event queue. Internal events are added as they occur. Next, the agent modifies the intention and desire structures by dropping all successful desires and satisfied intentions as well as impossible desires and unfeasible intentions."

Procedure 1: The BDI interpreter cycle

```
1  initialize-state();
2  repeat
3      options:= option-generator (event-queue);
4      selected-options:= deliberate (options);
5      update-intentions (selected-options);
6      execute();
7      get-new-external-events();
8      drop-unsuccessful-attitudes();
9      drop-impossible-attitudes();
10 end repeat
```

Among the critiques faced by BDI, and other deliberative architectures, are the difficulties to obtain solutions fast enough to face dynamic and complex environments that sometimes can only be managed by choosing suboptimal actions due to real-time restrictions.

5.2.2 *Reactive Agents*

The inherent complexity of symbolic representations in deliberative agents, and the difficulties to meet real time constraints have conducted researchers to analyze other agent architectural alternatives based in simpler models and bottom-up approaches.

Among them, the most popular has been the subsumption architecture [3] by Rodney Brooks, based on the hypothesis that intelligence is an emergent property on certain complex systems, and that there is no need to include symbolic models to obtain it. Subsumption architectures usually are organized in hierarchical layers (see Fig. 5.1), from lower to higher abstraction levels [3], that decompose the complete behavior of the agent into a set of sub-behaviors. Within this architecture, higher levels subsume lower ones in order to obtain a feasible practical behavior.

The major application of reactive agents, and subsumption architecture, has been in robotics. The reason is that robots need to model the complex inputs coming from the outside environment, and they must usually act fulfilling real time constraints, which become difficult for deliberative architectures and need more agile approaches. Reactive architectures only consider a set of basic actions, sometimes denoted as instincts or behaviors, and the problem is to decide which one should be executed next.

Typically the problems with these architectures appear when we face complex problems with a huge search space, and becomes difficult to the inherent short time view of the agent behavior. Besides, learning from the environment and from past actions are not naturally considered within these models.

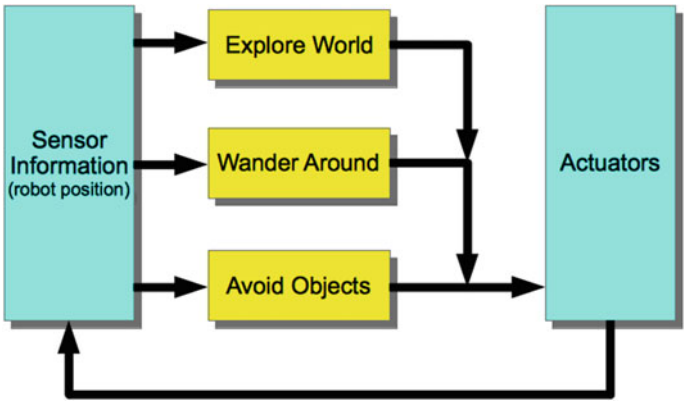

Fig. 5.1 An implementation of the subsumption architecture for robot navigation

5.2.3 *Hybrid Agents*

Due to the limitations, faced by deliberative and reactive architectures; a new type of architectures was considered to combine the advantages of both models, but trying to avoid their inherent limitations. These hybrid architectures typically consider a set of vertical or horizontal layers. Some of these layers work under a reactive model, for simple or urgent events that do not need complex reasoning, i.e., imitating animal instincts. Other layers work using a deliberative architecture, providing a more abstract model to generate medium and long term plans.

In a similar way to the Brooks' subsumption architecture described before, the layers are organized hierarchically over several abstraction levels, which in most of the models are three defining:

1. A reactive level, at the lower layer, to react to inputs in real time and that it is usually implemented using a subsumption architecture.
2. An intermediate knowledge level, describing the knowledge that the agent has about the problem and its environment, usually modeled with a deliberative architecture.
3. An upper social level, that manages the interactions with other agents in order to cooperate, compete or interact in a general term.

The two basic hybrid architecture models are horizontally layered, where all the layers have access to external devices; and vertical layered, where there is only a layer that has access to external sensor and actuator devices.

Horizontally Layered

The most typical example of an horizontal hybrid architecture is TouringMachines [9], where all the layers receive inputs from the external perception sensors; and each layer suggest what to do next. Therefore, there is a need for a control subsystem to take decisions about inhibiting certain inputs, deciding what layer has a temporal control over the agent, or deciding what output is selected (see Fig. 5.2). This architecture considers these three basic layers: a reactive layer, a planning layer and a modeling layer to describe the world, and to provide new goals for the planning layer.

Vertically Layered

The most popular example of a vertical layered architecture is InteRRaP [16], that is described in Fig. 5.3 and contains a reactive layer, a planning layer and a social layer to interact with other agents in the environment. As shown in the figure, the main difference is the interaction between the layers and the external systems, which in this case implies that all sensor inputs are managed vertically by each layer before feeding the upper one. The same happens with the action outputs, that are submitted by upper layers to lower ones to reach the external interface. Therefore, the information here first flows down-top, and then top-down when interacting with the environment.

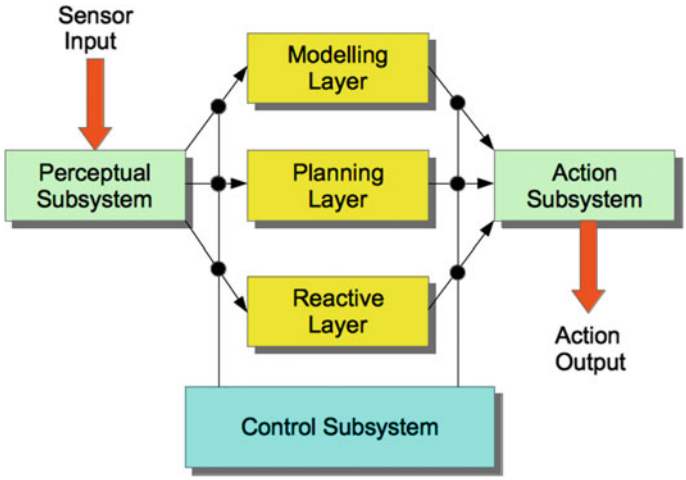

Fig. 5.2 TouringMachines: a horizontally layered agent architecture

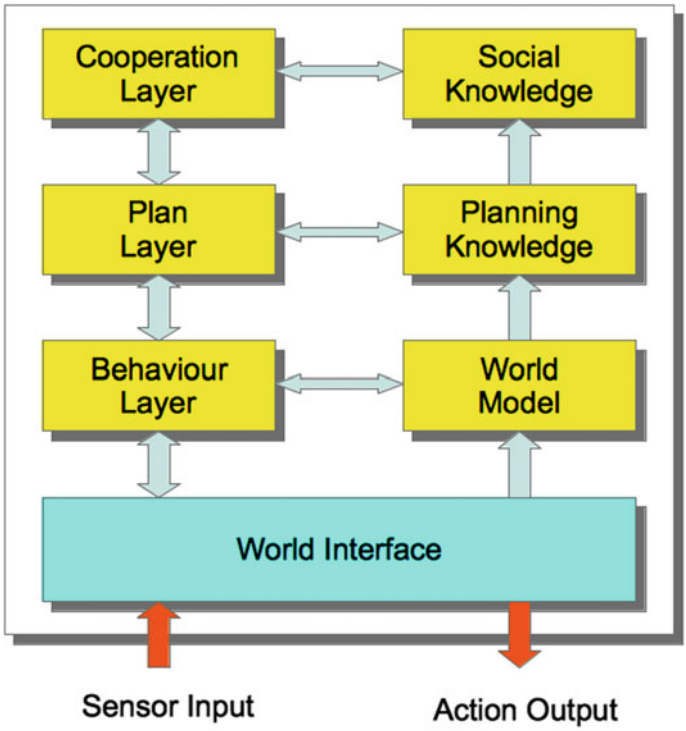

Fig. 5.3 InteRRap: a vertically layered agent architecture

5.2.4 Multi-agent Architectures

Due to its innate characteristics, multi-agent systems were born as a part of distributed artificial intelligence with the aim to solve complex problems or to simulate physical or virtual models. Therefore, we can not think about agents as isolated entities, and we have to consider them in relation to other agents, entities or humans with whom they communicate.

The possibility to consider multiple autonomous agents MAS is very suitable for complex and distributed problems, probably including complex dynamics, that are very difficult to be solved by classical centrally programmed entities. As a distributed approach, a multi-agent system usually has characteristics like:

- It usually contains a set of heterogeneous agents with their own skills, data and abilities for acting and interacting.
- The agents usually share a common objective, and the whole system must be able to divide the tasks to be performed by the agents, taking into account their abilities and processing capabilities.
- Each agent in the system has a limited view of the whole problem, the environment and the capabilities of the other agents. Sometimes this is referred as agents have limited resources (bounded rationality, computing time, memory storage, etc.), that they must combine in order to solve the whole problem.

Under this perspective appears the need to describe or set up the conditions, and the global architecture, where the multi-agent system must work, which normally implies the communication among the agents, their cooperation or competition. We will introduce in detail these concepts in the next section, but before we will describe the FIPA multi-agent architecture.

FIPA Architecture

FIPA (Foundation for Intelligent Physical Agents) architecture [12] has mainly work in solving interoperability among agents, and the development of new extensions to the architecture within a standardized framework. The most popular implementation of those standards has become JADE[1] (Java Development Framework).

FIPA mainly has described the external behavior of agents, while the internal decisions become part of their developers, and such external behavior follows an open model in order to help the interaction among multiple heterogeneous agents or agent societies. Then, the FIPA model mainly deals with the creation, deletion, registering, location and communication among agents.

The agent platform is the core of FIPA's reference model, and provides the infrastructure for the agent development and their use, including: the operating system, communications, middleware and agent management. The FIPA standard defines the services that must provide any agent platform like:

[1] See http://jade.tilab.com.

- *Agent Management System (AMS)*: the AMS is the main managing element concerning the platform and the agents it contains. It offers services for creation, deletion, state control, registration, mobility, sharing resources and communication. Besides it also provides a domain name service (Agent Name Service, ANS), also known as white pages. The AMS allows to control the agents' life cycle, and every agent may be: started, active, suspended or in stand-by.
- *Directory Facilitator (DF)*: the DF is a complement to the ANS, and in this case it provides a yellow pages service; allowing to search agents with certain skills or characteristics.
- *Message Transport System (MTS)*: all FIPA agents have access to the MTS, that supports communications between agents in the local platform or between the local and remote platforms. Agents communicate among them using the Agent Communication Language (ACL), which is another FIPA standard. The message communication model is asynchronous, and use queues for sending and receiving messages, with their own management policies.

Security is a relevant concept in any distributed system, and even more using multi-agent systems, given their autonomy and the complexity that the system may achieve. FIPA has considered several policies to maintain high levels of security within such open platforms. FIPA also has recommended existing standards whenever possible, like X.509 for public key infrastructures, or SSL (Secure Sockets Layer) among others.

Mobility is another service considered in the FIPA platform, and the standard tries to provide interoperability solutions for mobile agents traveling between different platforms. This mobility can consider cloning and running the local agent in a remote platform, which is the simplest case; or migrating the agent among different platforms, which is a more complex execution model. In the migration case, the agent is suspended in the local platform and sent over the network, with its code and data, to continue its execution on the remote platform.

The FIPA platform also provides an Ontology service to support communication among agents using ACL messages. This is helpful to allow agents to understand each other. To use this service, when an agent registers at the DF it informs about the ontologies it can manage, so other agents may know in advance how to communicate with him. The main functions of the ontology service are:

- Keeps a set of ontologies for public use, available for the agents.
- Translate expressions between different ontologies.
- Answers queries about terms of the managed ontologies.
- Helps to identify and to use shared ontologies among agents.
- Discovers new ontologies and make them available for the agents.

5.2.5 Mobile Agents

Mobile agents are a particular, and time ago popular, type of agents with the capability to decide when and where to move themselves (their code, state and data) between several devices connected through a communication network, in order to resume their previous execution at every new device. Therefore, besides the classical properties any agent may enjoy, mobile agents add the mobility feature to their profile.

The idea behind any mobile agent technology is, basically, to substitute remote procedure calls, between a client and a server, and execute the code directly at the server side gaining efficiency and reducing bandwidth consumption. Remote procedure calls invoke a procedure (or method in the OO terminology) on another process, which usually is remotely located. Normally, remote procedure calls block the client, while waiting for the answer of the server, i.e., they usually follow a synchronous communication model. An alternative, in the mobile agent paradigm, is to send the agent, i.e., the client process itself, to perform the task(s) directly on the server side and return back the results. Potentially, this could reduce significantly the bandwidth used in the classical client-server remote procedure call paradigm, and obtain an overall improvement of network and computing resources. Consider, for example, a classical search process done by any web spider along the Internet day after day. Classical web spiders request and index web pages, causing a big redundancy in the data sent by the web servers to the web search machines. It would be much more efficient to just obtain what has been modified since the last search. So, instead of requesting all the information and analyzing it at the web search machines, it would be faster to send an agent to execute its code in the web servers, and to detect what has been changed since the last visit.

There are a number of challenges to support mobile agent technology like: how to serialize all the agent's code and data to continue the execution on the remote device, how to execute the code if running on a different hardware or operating system, or how to protect both, the host and the agent from malicious attacks. There have been a number of attempts to try to minimize these difficulties. For the most cases, even not all of them, the use of the Java language has simplified (but not definitively solved) these difficulties, and has been adopted by a number of mobile agent platforms and solutions. The main reasons are that Java includes the serialization process within its own features; uses a Java Virtual Machine (JVM), so the code potentially can run in any device capable of running a JVM; and, finally, security has been one of the basis from where Java was developed. Nevertheless, a huge problem deals with security. The idea of allowing the execution of external code in the network devices seems potentially insecure, even if you trust in the network agents and nodes. Besides, it is very difficult to protect the spreading of malware, once the agent is attacked by a malicious host; which could have been infected previously.

Along the 90s, a set of mobile agent platforms were developed, being Aglets[2] the most popular mobile agent framework. Aglets takes advantage of the Java technology to overcome some of the challenges described above.

[2] http://aglets.sourceforge.net/.

5.3 Ontologies

What is the aim of communication and how it can influence communicating entities and the external environment? These questions have been addressed along the second half of the XX century by analytical philosophers like Austin [1] and Searle [22], which drive to the study and classification of *speech acts*. Their reflections have been reused by the agent community when trying to communicate different agents.

The problem of communicating two different entities make necessary the establishment of some agreement between both parts about the terminology they will use to describe the domain in order to effectively exchange knowledge. In order to solve such problem, an *ontology* specifies a set of terms that must provide a basis for understanding each other on a certain domain. Ontologies have arisen and become popular in the last fifteen years due to the emergence of the *Semantic Web*. The idea was to enrich web pages with semantic information in order to simplify how the information is processed by computers.

There have been a number of languages used for creating ontologies. Now we enumerate a few of them, which became popular:

- *RDF*: the Resource Definition Framework is not an ontology language, but very closely connected with ontologies and the semantic web. RDF was developed with the idea of providing a framework to represent knowledge in web pages or resources. RDF is very simple, and this makes things easier in general, but sometimes limited in expressive power. RDF basically represents subject-predicate-object triples describing knowledge about its elements, denoted as resources, and is based in XML. Another language related with RDF is RDF-Schema (RDFS), which is a simplified set of resources and standardized properties to create RDF vocabularies, and, for instance, to check out if a set of RDF triples is valid (p.e., a clock has not surnames) and the linked values and their types are also valid (p.e., the price of a product is not "sunny"). As XMLS models syntactic relations among data, RDFS models semantic relations, i.e., knowledge. But RDFS only allows to define very simple ontologies, so more complex and powerful languages have been defined for such purposes.
- *OWL*: the Web Ontology Language is presently the most relevant language for writing ontologies. There are three main levels of OWL: *OWL Lite* (which is the simplest and less expressive version, but easier to manage computationally and to understand by humans), *OWL DL* (which extends OWL Lite to include *description logic*) and, finally, *OWL Full* (which is a very expressive and powerful framework, but much more difficult for checking out consistency properties).
- *KIF*: the Knowledge Interchange Format is closely related with first-order logic, that emerged in the agents area, as a difference with the three previous languages mentioned before. KIF allows agents to express properties of objects, relationships between objects and general properties in a domain. KIF uses first-order logic elements, such as the Boolean connectors (*and, or, not*) and the universal and existential quantifiers *for all* and *exists*. KIF provides a basic vocabulary of objects (numbers, characters, and strings) and some standard functions and rela-

tions among objects (e.g., *less than* or *addition* for numbers). KIF uses a LISP-like notation for handling list of objects.

A set of software tools have been developed in the past years to design and refine ontologies. Among them, the most popular probably are Ontolingua server [8] and Protégé [19]. Ontolingua is a web-based service intended to provide a common platform to share and unify ontologies among different groups. The main component is a library of ontologies, expressed in the Ontolingua ontology definition language (based on KIF). Protégé has been a very popular tool for ontology development, and uses a platform-independent editor for creating class hierarchies, properties and instances that can be imported or exported in several formats, including OWL.

5.4 Communication

The theory of speech acts is based in the seminal work of the philosopher John Austin [1], as he reflected that a certain class natural language expressions (*utterances*) could influence the world in a similar way of physical actions do. He denoted such utterances as *speech acts* and examples could be the word *yes* when accepting a marriage or a confession in front of a jury. Then, Austin denoted the act of speech as a set of *performatives*, which refer to a set of contents that could be true or false, but also could express intentions, decisions, promises, etc.; which are not necessarily true or false. Examples of such performative verbs are: request, inform and promise. Austin also distinguishes three different aspects of speech acts: locution (the propositional content, e.g., 'please, give me the key'), illocution (the action that the performative does, e.g., 'she requested me to give her the key'), and perlocution (the effect really caused on the receiver, e.g., 'I gave her the key').

John Searle extended Austin's work in the book *Speech Acts* [22], where he identified several properties necessary for a speech act to succeed. These conditions are:

- *Normal I/O conditions*: state the speaker and the hearer are under normal circumstances, p.e., the speaker is not dumb, the hearer is not deaf, and the environment in not so much noisy.
- *Preparatory conditions*: state that the hearer must be able to perform the action potentially requested by the speaker.
- *Sincerity conditions*: that describe if the speaker really wants the action to be performed.

Searle also identified five types of speech acts:

- *Assertives*: commits a speaker to the truth of the expressed proposition (p.e., informs).
- *Directives*: cause the hearer to take a particular action (p.e., request, commands, etc.).

- *Commissives*: commits the speaker to do some future action (p.e., promises).
- *Expressives*: express the speaker emotions towards the proposition (p.e., congratulations, excuses, thanks, etc.).
- *Declaratives*: change the reality in accordance with the proposition (p.e., baptisms, guilty sentence, marriage acceptance, etc.).

The technical group *Knowledge Sharing Effort* from DARPA (Defense Advanced Research Projects Agency) worked in standardizing a communicating language among agents in the early nineties. The main outcome was KQML (Knowledge Query and Manipulation Language)[10], a message-based language for agent communication. KQML is both a message format and a message-handling protocol to support run-time knowledge sharing among agents. KQML can be used to interact with an intelligent system, or by two or more intelligent systems to share knowledge among them. KQML contains an extensible set of performatives, which comprise a substrate on which to develop higher-level models of agent interaction such as contract nets and negotiation, and can be divided in three main classes: discourse, intervention, and network communication. Several critics has been done to this language, especially concerning its interoperability problems among different implementations, the lack or rigorous semantics, the absence of performatives of type commissive and a performative set too large [27].

Nevertheless, KQML was a relevant pillar for FIPA to develop its own Agent Communicative Language (ACL), which defines the structure that must have a message exchanged among agents in a network, and was defined having in mind all the critics done to KQML. ACL, which looks very similar to KQML, but with a reduced set of performatives, and it was defined considering a comprehensive formal semantics. The specification consists of a set of message types, and the description of their pragmatics, that is the effects on the mental attitudes of the sender and receiver agents. Every communicative act is described with both a narrative form and a formal semantics based on modal logic. The specification also provides the normative description of a set of high-level interaction protocols and several kinds of auctions. Find next an intuitive example taken from [11]:

```
(ask-if
 :sender I
 :receiver j
 :content (= (weather England (summer 1997)) wet)
 :ontology meteorology
 :reply-with query-17)

(inform
 :sender j
 :receiver I
 :content true
 :in-reply-to query-17)
```

Basically agent *I* ask to agent *j* if the weather was wet in England during the summer of 1997. Agent *j* answers that the answer is *true*, i.e., it was wet indeed. FIPA ACL has been considered as part of FIPA-OS.

5.5 Coordination

As we have seen in previous sections Multi-agent Systems are usually composed by multiple computing entities interacting in an autonomous and sometimes intelligent way. In order to manage appropriately the resulting complex system, the capability to coordinate with other agents, implicitly or explicitly, is a key element of any agent.

Given a particular problem, or a global task to be solved, once agents have agreed with respect to a common goal, it is needed to specify the details of the collaboration. For instance, providing the tasks that must be performed by every one, when they must be finished, what to do if problems appear, when one or several particular tasks have failed, etc.

One relevant tool to coordinate the agents is the use of Distributed Problem Solving (DPS) [5], that usually assume that agents are benevolent and cooperative, i.e., not necessarily self-interested. This means that they do not try to maximize their own utility function, but the whole MAS maximizes the global one. Usually this condition is accepted assuming that the whole MAS is built by the same designer or, more realistically, that self-interested agents have reached a previous agreement. The use of DPS has a set of advantages:

- It takes advantage of specialized agents, and allows to distribute the tasks depending on the characteristics, load, and skills of the agents in the system.
- This distribution allows to take decisions locally, avoiding to overload the networks with inefficient managing information.
- Besides it allows agents to be physically distributed, taking advance of several computing infrastructures; and to use parallel computing, that can enhance the performance depending on the system characteristics and the tasks to be done.

Under this approach, when an agent must perform a certain set of tasks it can search other agents to ask them for help. Usually, task decomposition has a number of steps like: dividing a big task into subtasks, finding the appropriate agents to perform those subtasks, and gathering the results from the subtasks to get the original task solved. Note that this problem decomposition can be performed recursively by any of the agents participating in the process.

If the systems is homogeneous, then task decomposition becomes trivial among the agents, and only will take into account their availability, usually depending on their overload at a certain moment. The problem becomes more difficult, but interesting, when the agents are heterogeneous and have different skills that allow them to perform the different tasks more or less efficiently.

One classical approach to assign subtasks in a MAS is the Contract Net Protocol (CNP) [23]. In CNP there is an agent that behaves as the manager, and it is the

responsible for dividing a task into a set of subtasks, and for finding the appropriate agents to solve the global tasks under a required deadline. Usually the manager broadcast the subtasks to the set of available agents, consider their bids to solve the subtasks; and, depending on the offers, perform the best possible assignment.

Sometimes, when accomplishing a certain task, it is useful to distribute the results among several agents in order to improve the global performance. This usually happens when several agents face the same problem, and they exchange their approach and results in a certain agent neighborhood, so they can increase their confidence in their own solutions. At the same time, an agent can use other agents' solutions to refine its particular approach. This can be done, distributing the information among the interested agents ad-hoc, or creating some kind of structures to adequately organize and store the relevant information by a coordinator or by repositories. Nevertheless, the sharing of results should be done if the agents know how to adequately integrate other agents context and results. Besides, the communication level among the agents must be kept under a certain threshold, avoiding to exchange big amounts of information that can use too much network bandwidth.

In a MAS system, where agents collaborate in a distributed problem solving framework, it is usually needed to conceive a global plan, that considers the actions for every agent, taking into account other agents' actions too. This is known as distribute planning, and it can be designed from three points of view:

1. The system can do a central planning, that afterwards is distributed among the agents, and then they execute their own sub-plan distributively.
2. The different agents in the system can create distributed and partial sub-plans, that afterwards are composed and organized, by a set of organization agents, to create a global plan for the whole system.
3. We also can consider the creation of distributed sub-plans, that must be somehow revised to avoid conflicts, and then will be implemented in a distributed way.

One classical planning framework is the Partial Global Planning [6], which mainly works as follows:

- An agent analyzed and decompose its own tasks to create a local plan with its main objectives.
- The local plan is abstracted, to its main elements, and send to other related agents to check its common goals and conflicting elements.
- Local plans can be integrated into a global plan, trying to improve coordination and avoiding conflicts or redundancies.
- Then, it is decided how to communicate the agents to provide the results of their own tasks.
- Afterwards, the local plans are provided to the local agents to be performed.
- During the execution of the plan, the results and unexpected problems are considered, in order to decide when to do a re-planning. The idea is to avoid that minor changes cause too many plan changes or even a complete reorganization.
- Also, if some agents become overloaded, they can try to get help from idle agents, changing a bit the organization to load distribution.

5.6 Methodologies

As commonly happen with Software Engineering technologies, there is an increasing effort to support the development of new systems using them. Developments in the framework of Multi-agent Systems have not been an exception. Methodologies for the analysis and design of agent-based systems fall into two main groups: the ones that take their models from object-oriented development, adapting OO methodologies to the purposes of Agent-oriented Software Engineering (AOSE); and the rest of them that adapt knowledge engineering or alternative techniques.

In general, the most common approach for developing methodologies belong to the first category, probably because it is highly common to have a previous background in object-oriented development. Nevertheless, there are some remarkable differences between agents and objects [27], for instance proactivity in agents systems, or the cooperation or negotiation among self-interested agents. Several approaches have been extended in order to improve those deficiencies, but the development of multi-agent systems using formal methodologies has not became yet mainstream.

Some of the most popular methodologies include:

- The Object Management Group [25] and FIPA collaborated to support the development of notations from object oriented design for modeling agent-based systems. Users with experience with UML (Unified Modeling Language) and object oriented techniques, can find in MaSE (Multi-agent Systems Engineering, [4]) a possibility that greatly reduces the learning curve.
- The AAII methodology used the PRS-based Belief-Desire-Intention technology, and the distributed multiagent reasoning system (DMARS) [13]. This methodology is also based on an object-oriented background, enhanced with some agent-based concepts.
- Gaia [29] takes its roots from object-oriented analysis and design, but it also provides a set of agent concepts to support agent-oriented development. Mainly, Gaia encourages the developer to construct agent-based systems as a continuous process of *organizational design*, where each refining step gets systematically closer to the final implementation.
- Tropos [2] provides a conceptual framework for modeling agent-oriented systems based on the development of an *actor model* and a *dependency model*. The former models the key stakeholders and their goals, while the latter models the dependencies among actors and resources. Then a *goal model* is developed to detail those particular goals, and finally a *plan model* to describe how to achieve them.
- The Prometheus methodology [17] consists on three main stages: (i) a *system specification* to identify the goals, the basic functionalities and the system interfaces; (ii) an *architectural design* to identify the agent types, the system structure and the interaction among the agents; and, finally, (iii) a *detailed design* to refine agents capabilities and detail the system processes.
- MAS-CommonKADS [14], is another popular methodology especially for those designers familiarized with knowledge engineering systems.

- INGENIAS [18] became a great alternative methodology for modeling properties closer to the agent world paradigm, like intentional properties or agent sociability.

5.7 Modeling and Simulating Complexity

Multi-agent systems and its associated models for processing and communication have become excellent tools to model and simulate complexity in Science. For instance, they have been used to analyze emergent properties, such as big structures emerging from simple local interactions, which lead to wonder how do natural systems self-organize by themselves to reach such high levels of complexity. At the same time, selfish agents can cooperate with others in order to create larger coalitions that drive to a higher common wealth. All these ideas together with the aggregation (composition) and specialization (inheritance) properties of multi-agent systems are very interesting for analyzing and understanding natural systems from a scientific point of view, and for creating new efficient ones, from an engineering perspective.

5.8 Conclusion

Agent-based simulations have become an interesting framework to model and explore complex systems, as they support the idea that the global behavior of a multi-agent system derives from low-level interactions among its composing agents. The fundamental element of many models to describe complex adaptive systems is the adaptive agent, which is an entity that uses its experience in a changing environment to continually improve its abilities and to reach its goals.

This chapter has introduced the concept of agents, as the interacting entities that may compose a complex system. We have seem several types of agent architectures, from deliberative to reactive ones, and their composition as hybrid agents. From such agent definitions, the concept of multi-agent system has been derived. Then, we have introduced the use of ontologies to effectively support the exchange of knowledge among the agents. We also have seen several approaches to coordinate and plan the actions in a multi-agent system. Then, several methodologies for agent-oriented engineering have been described. Finally, the potential of the multi-agent model to simulate complex systems has been addressed.

The next chapters extend the possibilities of multi-agent systems to describe frameworks where agents play a fundamental role. Chapter 6 describe scenarios, where a multi-agent system behaves as a self-organized global entity, with possible emergent properties and behaviors, not explicitly present in the individual agents. Chapter 7 is the last one of this background, and introduces Game Theory as a framework where agents collaborate or compete depending on their own interest and the type of problem or game they have to face.

5.9 Further Reading

The interesting reader can get a nice and overall introduction to the topic at the book *Introduction to Multiagent Systems* [27] by Mike Woorldridge. A more technical book is *Multiagent Systems* [26], written by several authors leading their respective fields, and edited by Gerhard Weiss.

References

1. Austin, J.L.: How to Do Things with Words. Harvard University Press, Cambridge (Massachusetts) (1962)
2. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: an agent-oriented software development methodology. Auton. agents multi-agent Syst. **8**(3), 203–236 (2004)
3. Brooks, R.A.: Intelligence without representation. Artif. Intell. **47**, 139–159 (1991)
4. Deloach, S.A., Wood, M.F., Sparkman, C.H.: Multiagent systems engineering. Int. J. Softw. Eng. Knowl. Eng. **11**, 231 (2001)
5. Distributed problem solving and planning. Chapter in the book Multi-agent Systems and Applications. Lecture notes in computer science, pp. 118–149 (2001)
6. Durfee, E.H., Lesser, V.R.: Partial global planning: a coordination framework for distributed hypothesis formation. Syst. Man Cybern. IEEE Trans. **21**(5), 1167–1183 (1991)
7. Erman, L.D., Lesser, V.R.: A multi-level organization for problem solving using many diverse cooperating sources of knowledge. In: Advanced Papers of the Fourth International Joint Conference on Artificial Intelligence, Tiblisi, Georgia (URSS) (1975)
8. Farquhar, A., Fikes, R., Rice, J.: The Ontolingua server. Int. J. Hum. Comput. Stud. **46**(6), 707–727 (1997)
9. Ferguson, I.A.: TouringMachines: an architecture for dynamic, rational, mobile agents. Doctoral dissertation, University of Cambridge (1992)
10. Finin, T., Fritzson, R., McKay, D., McEntire, R.: KQML as an agent communication language. In: Proceedings of the third International Conference on Information and Knowledge Management, ACM (1994)
11. FIPA ACL Message Structure Specification. http://www.fipa.org
12. Foundation for Intelligent Physical Agents. IEEE Computer Society standards organization. http://www.fipa.org
13. Georgeff, M.P.: Distributed multi-agent reasoning systems (dmars). Technical report, Australian AI Institute, Level 6, 171 La Trobe Street, Melbourne, Australia (1994)
14. Iglesias, C.A., Garijo, M., Gonzlez, J.C., Velasco, J.R.: Analysis and design of multiagent systems using MAS-CommonKADS. Intelligent Agents IV Agent Theories, Architectures, and Languages. Lecture Notes in Computer Science, vol. 1365, pp. 313–327 (1998)
15. Lesser, V.R., Corkill, D.D.: Functionally accurate, cooperative distributed systems. IEEE Trans. Syst. Man Cybern. SMC **11**(1), 81–96 (1981)
16. Müller, J.P.: The Design of Intelligent Agents: A Layered Approach. Springer Science & Business Media, New York (1996)
17. Padgham, L., Winikoff, M.: Prometheus: a methodology for developing intelligent agents. Agent-oriented Software Engineering, pp. 174–185. Springer, Berlin (2002)
18. Pavón, J., Gómez-Sanz, J.: Agent oriented software engineering with INGENIAS. Multi-agent Systems and Applications III. Lecture Notes in Computer Science, vol. 2691, pp. 394-403. Springer, Berlin (2003)
19. Protégé Website. http://protege.stanford.edu/
20. Rao, A.S., Georgeff, M.P.: BDI Agents from Theory to Practice. In: Proceedings of the First International Conference on Multi-agent Systems (ICMAS-95) (1995)

21. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach, 3rd edn. Prentice Hall, New Jersey (2014)
22. Searle, J.R.: Speech Acts: An Essay in the Philosophy of Language. Cambridge University Press, Cambridge (1969)
23. Smith, R.G.: The Contract net protocol: high-level communication and control in a distributed problem solver. IEEE Trans. Comput. **29**(12), 1104–1113 (1980)
24. Sycara, K.: Multiagent systems. AI Mag. **10**(2), 79–93 (1998)
25. The Object Management Group. http://www.omg.org
26. Weiss, G.: Multiagent Systems, 2nd edn. MIT Press, Cambridge (2013)
27. Wooldridge, M.: Introduction to Multiagent Systems, 2nd edn. Wiley, New York, NY, USA (2009)
28. Wooldridge, M., Jennings, N.R.: Intelligent agents: theory and practice. Knowl. Eng. Rev. **10**(2), 115–152 (1995)
29. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Developing multiagent systems: the Gaia methodology. ACM Trans. Softw. Eng. Methodol. (TOSEM) **12**(3), 317–370 (2003)