

Agents, Multi-agent Systems and Mobile Code

2.1 Overview of Agent Technology

While there is no fixed definition of an *agent* (or *software agent*), the concept is typically used to refer to software components that have their own thread of control (and hence may act autonomously), and are capable of sensing and reacting to changes in some environment. Often software agents have other properties, such as the ability to communicate with other agents. Recently, software agents have become widely used in the modelling of complex, distributed, problems [7]. This section discusses in more detail the various types of agents that are in use, along with multi-agent systems, which are systems in which agents interact in order to solve some problem or achieve a set of goals, and mobile agents, which are agents capable of moving from one server to another during their execution.

2.2 Intelligent/Autonomous Agents

2.2.1 Definitions

There are a number of definitions of an *intelligent agent*. One of the more widely used is that put forward by Wooldridge and Jennings [29], which defines an agent as a system that “enjoys the following properties:

- autonomy: agents operate without the direct intervention of humans or others, 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 (which may be the physical world, a user via a graphical user interface, a collection of other agents, the Internet, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it;

- pro-activeness: agents do not simply act in response to their environment they are able to exhibit goal-directed behaviour by taking the initiative.”

In the same paper, Wooldridge and Jennings described the notion of a *strong agent*, used by researchers in the artificial intelligence field, which is an agent that “is either conceptualised or implemented using concepts that are more usually applied to humans”. An example of a strong agent is one based on a *mental state* described in terms of beliefs, desires, intentions and commitments. In contrast to the weak notion, the characteristics of the *strong agent* are listed below:

- clearly identifiable problem-solving entities with well-defined boundaries and interfaces;
- situated in an environment, agents perceive through sensors and act through effectors;
- designed to fulfil a specific purpose, the agent has particular objectives to achieve;
- autonomous, the agent has the capability of control, the effectors depend on both its internal states and behaviours;
- capable of exhibiting the problem-solving behaviours in pursuit of its design objectives. The agent needs to be both reactive (able to respond in a timely fashion to changes that occur in its environment) and proactive (able to act in anticipation of future goals).

Strong agents can also be known as *cognitive* agents, in comparison to simpler *reactive agents*, which are agents that act only in response to changes in the environment [8]. One of the simplest type of reactive agent is an agent having a series of IF-THEN rules, mapping from input states to actions.

2.2.2 Intelligent Agents in Information Processing and Problem Solving

Over the past few years, a revolution has taken place in information generation and dissemination. Global distribution of information can now be made available via the Internet very easily. However, the problems of using the Internet and Web to retrieve information of interest are growing, and include:

- inexhaustible pool of information;
- distributed, heterogeneous nature of information and information services;
- difficulty of retrieving relevant information and more time spent searching for information.

Useful tools are needed for users to reactively and pro-actively search, retrieve, filter and present relevant information. Querying and integrating heterogeneous data from distributed sources has been a focus for research in recent years, as the notion of agent software assistants that help users to achieve

productivity gradually becomes the trend in network computing. If given “intelligence”, agents could be extremely useful, which means that agents can be trained to learn user actions and preferences, and perform appropriately to similar actions or specific preferences next time.

Intelligent software agents have obtained encouraging results in solving the problems of current (threat of) information overkill and information retrieval (IR) over the Internet. The complexity of information retrieval and management would be mitigated if intelligent means could be provided. Take “ACQUIRE” [30] for example, the following functions of information retrieval should perform in theory:

- goal-oriented: ACQUIRE is only looking for what the user wants, with consideration of how to most efficiently satisfy the request;
- integrated: ACQUIRE provides a single, expressive, and uniform domain model from which queries may be posed;
- balanced: ACQUIRE attempts to balance the cost of finding information on its own using planning and traditional database query optimisation techniques.

Furthermore, the following three stages are implemented by ACQUIRE

- respond to a query from a user and decomposes it accordingly into a set of sub-queries with site and domain models of the distributed data stores;
- generate an optimised plan intelligently for retrieving answers to these sub-queries over the Internet and deploys a set of intelligent mobile agents to delegate these tasks;
- merge the answers returned by the mobile agents and then return them to the user.

An example of agent coordination for problem solving is the Reusable Task Structure-based Intelligent Network Agents (RETSINA) [31]. The multi-agent-based RETSINA infrastructure has been developed at the Carnegie Mellon University in Pittsburgh, USA.

The RETSINA framework has been implemented in Java programming language. It is being developed for distributed collections of intelligent software agents in problem solving. The system consists of three types of reusable agents, which can be adapted to address a variety of domain-specific problems. Three major intelligent agents are integrated into the system architecture:

- an interface agent for interacting with users to receive the requirements and display the outcomes;
- a task agent, as an assistant to the users, which performs tasks according to the problem-solving plans, and queries and exchanges information with other agents;
- an information resource agent for intelligent access to a heterogeneous collection of information sources.

2.3 Agent Architectures

Four basic agent architecture categories and key examples are reviewed in this section; deliberative architectures, reactive architectures, learning-based architectures and layered architectures.

2.3.1 Deliberative Architectures

There are several different architectures for intelligent agent implementation. A well-known cognitive architecture is the belief-desire-intention (BDI) architecture [32, 33], in which the agent's knowledge base is described by a set of *beliefs* (those facts which an agent considers to be true), *desires* (those conditions which the agent wishes to bring about), and *intentions* (actions which the agent has committed to perform). These are explicitly represented in the knowledge base; for example, the Procedural Reasoning System (PRS) implementation [33] represents beliefs and goals as ground literals (sentences containing no implications, binary operators or variables) in first-order logic [34]. As described in [34], a BDI agent is capable of both reactive and deliberative behaviour. On each execution cycle of the interpreter, the agent retrieves new events from the environment. It then generates a set of *options*, which are plans or procedures that the agent is capable of carrying out, both in response to events and in order to achieve its goals. The agent will then execute, or partially execute, one or more of the selected options. This process is repeated for the agent's lifetime.

The BDI architecture was originally developed in the early 1990s, based on a model of human reasoning developed by Michael Bratman. One of its most famous early implementation was the PRS system, developed by Ingrand, Rao and Georgeff [33]. Rao and Georgeff, among others, have published a large number of papers concerning the BDI architecture and its implementation. The basic structure of a BDI agent is shown in Figure 2.1.

BDI Algorithms

Singh, Rao and Georgeff [34] give a set of algorithms for a basic BDI interpreter. The main interpreter loop is as follows:

```
BDI-Interpreter()
initialise-state();
do
  options := option-generator(event-queue, B, G, I);
  selected-options := deliberate(options, B, G, I);
  update-intentions(selected-options, I);
  execute(I);
  get-new-external-events();
  drop-successful-attitudes(B, G, I);
```

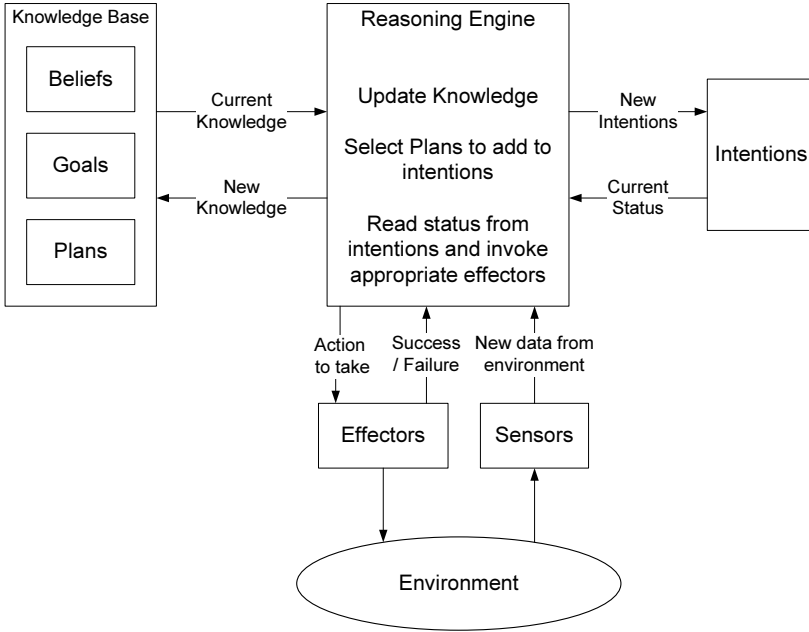


Fig. 2.1. Structure of a BDI agent

```

drop-impossible-attitudes(B, G, I);
until quit;

```

In addition, the option generation procedure is defined as follows:

```

option-generator(trigger-events)
options := {};
for trigger-event ∈ trigger-events do
  for plan ∈ plan-library do
    if matches(invocation(plan), trigger-event) then
      if provable(precondition(plan), B) then
        options := options ∪ {plan};
  return(options)

```

The deliberation procedure can be implemented in a number of ways. In the PRS system, a procedure was implemented that allowed the user to include a set of *metalevel plans* that were invoked in order to select other plans to be carried out. The deliberation procedure was then as follows:

```

deliberate(options)
if length(options) ≤ 1 then return (options);
else metalevel-options := option-generator(b-add(option-set

```

```

(options)));
selected-options := deliberate(metalevel-options);
if null (selected-options) then
    return (random-choice(options));
else return (selected-options);

```

BDI Components

The mental state of a BDI agent has three main components: beliefs, desires (or goals) and intentions.

Beliefs

The beliefs of an agent are those things which the agent considers to be true. For example, I might believe that the circuit breaker “bolney/cb182” is open.

Singh, Rao and Georgeff [34] give a formal definition of BDI concepts based on modal logic. They state that:

- An agent can change its beliefs over time.
- If an agent believes a condition, then it believes that it believes that condition. If an agent does not believe a condition, it believes that it does not believe it.
- If an agent believes a condition, then it must not believe that it does not believe that condition.

In PRS and other implemented BDI systems [32], it is common to represent beliefs as *ground expressions* in predicate calculus, in order to reduce the execution time of the option generation and deliberation procedures. In this case, they consist only of a predicate symbol and a set of literals, and cannot contain variables or operators such as implication, conjunction and disjunction. Also, the beliefs represent only those facts that the agent *currently* believes. For example, the statement that the circuit breaker is open could be represented in the agent’s knowledge base as:

```
status(bolney/cb182, open)
```

For reasoning about the behaviour of agents, Singh, Rao and Georgeff [34] use a modal operator Bel, and a relation B. This formalism is also used by Cohen and Levesque [35]. Suppose that x is an agent, and that p is a proposition. Then

$\text{Bel } x \ p$

means that x believes p . FIPA SL [36] includes the B relation, which has a similar meaning:

$(B \ x \ p)$

Desires and Goals

The desires of an agent are conditions that the agent would like to become true. Desires may be inconsistent with each other, and it may not be possible for the agent to achieve all of its desires. The agent's goals are a subset of its desires, which must be consistent and achievable.

In actual implementations it is possible to simplify the agent by having only goals and no desires.

Cohen and Levesque [35] define two types of goal: achievement goals and maintenance goals. An agent having an achievement goal in relation to a condition believes that that condition is not currently true, and desires to bring it about. An agent having a maintenance goal in relation to a condition believes that that condition is currently true, and desires that it should remain true.

Intentions

The intentions of an agent are those goals that the agent has committed to achieving, or actions that the agent has committed to carrying out. The commitment that an agent makes to its intentions is not indefinite. If an agent decides that one of its intentions is impossible, or that the justification for that intention has been removed, then it may drop or revise that intention.

Cohen and Levesque [35] state that if an agent intends to achieve a condition p , then:

1. the agent believes that p is possible;
2. the agent does not believe that he will not bring about p ;
3. under certain circumstances, the agent believes that he will bring about p .

They also discuss whether or not an agent intends to bring about the expected side-effects of its intentions. They state that the usual view is that agents do not intend these side-effects, but that in their theory the side-effects are "chosen, but not intended".

2.3.2 Reactive Architectures

The subsumption architecture [37] is an example of a reactive architecture which does not employ an explicit knowledge representation. A subsumption agent consists of a number of concurrently-executing *behaviours* [38]. These are arranged in a number of layers, with lower layers representing simpler behaviours, which have a high priority, and higher layers representing more abstract behaviours, and having lower priority. Low-level behaviours are unaware of the presence of the high-level behaviours. It is therefore possible to construct an agent using the subsumption architecture starting with the lowest-level behaviour and working upwards, with the agent being functional,

at least to a certain extent, after each layer is constructed. For example, Brooks [37] describes a mobile robot with a number of layers, performing tasks such as “avoid objects” (the lowest layer), “wander”, *etc.*, up to “plan changes to the world” and “reason about behaviour of objects” (the highest layer).

2.3.3 Learning-based Architectures

Learning-based architectures, such as reinforcement learning [39], genetic programming [40], or inductive logic programming [41], may be used to enhance the performance of an agent. While it is possible to use learning to improve the capabilities of an agent using an architecture such as BDI (for example, [42] used machine learning methodologies to recognise plans being undertaken by other agents in a BDI architecture and [43] uses case-based reasoning in a BDI agent for information retrieval), it is also common to incorporate learning into a much simpler agent architecture. Learning agents have been applied in a number of domains, including user interfaces [44], telecommunications [45], control and robotics [46].

2.3.4 Layered Architectures

Layered architectures such as TouringMachines [47] and INTERRAP [48] are cognitive architectures consisting of one or more layers. According to Ferguson, the advantage of a layered architecture is that a layered agent, by having different levels of behaviour operating concurrently, is capable of reacting to changing circumstances while planning its future actions and reasoning about the behaviour of other agents. Both of the architectures mentioned have three layers: TouringMachines has a reactive layer, modelling layer and planning layer, while INTERRAP has a behaviour-based layer, local planning layer and cooperative planning layer. In TouringMachines, all three layers are connected to the agent’s sensors and effectors. The three layers operate concurrently and are unaware of each other, while a control mechanism is used to filter the inputs and outputs and prevent conflicts. In INTERRAP, the sensors and effectors are connected only to the lowest layer (the behaviour-based layer). Activation requests are passed upward through the layers, and commitments are passed downwards. Unlike the subsumption architecture (which is a form of layered architecture), both TouringMachines and INTERRAP are based on explicit knowledge representation [47].

2.4 Standards for Agent Development

2.4.1 Foundation for Intelligent Physical Agents Standards

Foundation for Intelligent Physical Agents (FIPA) is an international organisation that is dedicated to promoting the industry of intelligent agents by

openly developing specifications supporting inter-operability among agents and agent-based applications. FIPA standards was originally proposed in 1996 to form the specifications of software standards for heterogeneous and interacting agents and agent-based systems [49]. In the past a few years, FIPA has been widely recognised as the major standard in the area of agent-based computing. Many standard specifications have been developed, such as Agent Communication Language (ACL) and Interaction Protocols (IPs), *etc.* On 8 June 2005, FIPA was officially accepted by the IEEE Computer Society. Figure 2.2 shows the overview of the FIPA standards.

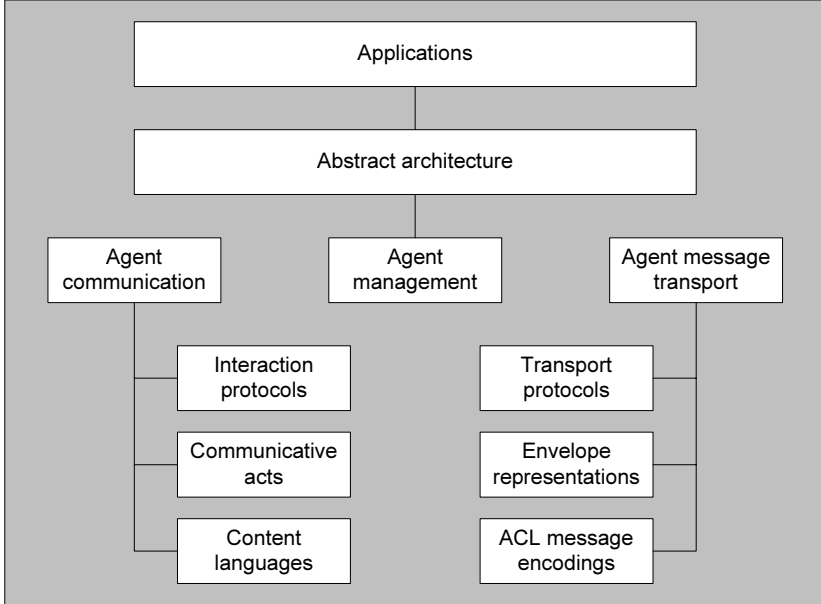


Fig. 2.2. Overview of the FIPA standards

FIPA Abstract Architecture

The FIPA Abstract Architecture specification (SC00001L)¹ acts as an overall description of the FIPA standards for developing multi-agent systems. The main focus of the FIPA Abstract Architecture is to develop semantic meaning message exchange between the different agents. It includes the management of multiple message transport and encoding schemes and locating agents and servers via directory services. Figure 2.3 demonstrates the FIPA Abstract Architecture mapped to different concrete realisations. In addition, it also

¹ All the specifications mentioned in this section are obtained from the website of FIPA organisation. <http://www.fipa.org/repository/standardspecs.html>

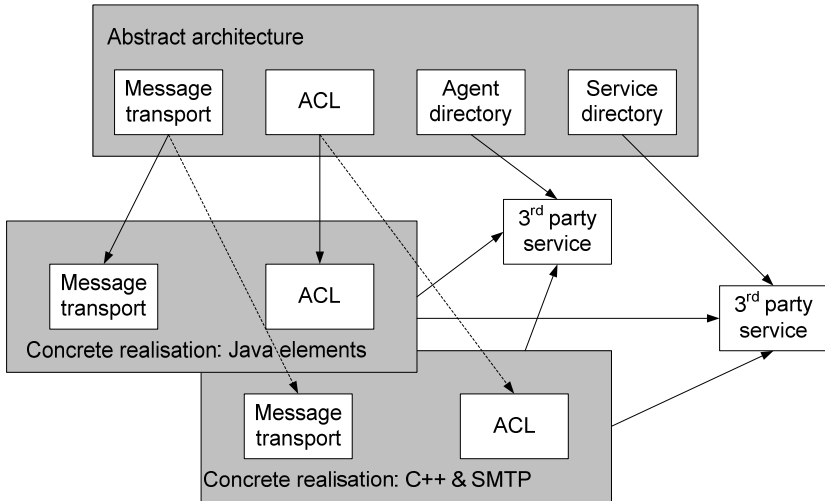


Fig. 2.3. FIPA abstract architecture mapped to different concrete realisations

supports mechanisms to create the multiple concrete realisations for interoperation. The scope of this architecture includes [50]:

- a model of services and discovery of services available to agents and other services;
- message transport interoperability;
- supporting various forms of ACL representations and content languages;
- supporting the representations of multiple directory services.

FIPA Agent Management System Standards

The FIPA Agent Management System specification (SC00023K) denotes an agent management reference model of the runtime environment that FIPA agents inhabit. The logical reference model is established for agent creation, registration, communication, location, migration and retirement [50]. The reference model includes a set of logical-based entities, such as:

- an agent runtime environment for defining the notion of agenthood used in FIPA and an agent lifecycle;
- an Agent Platform (AP) for deploying agents in a physical infrastructure;
- a Directory Facilitator (DF) which provides a yellow pages service for the agents registered on the platform;
- an Agent Management System (AMS) acting as a white pages service for supervisory control over access to the agent platform;
- a Message Transport Service (MTS) for communication between the agents registered on different platforms.

Figure 2.4 gives the FIPA agent management reference model constitution.

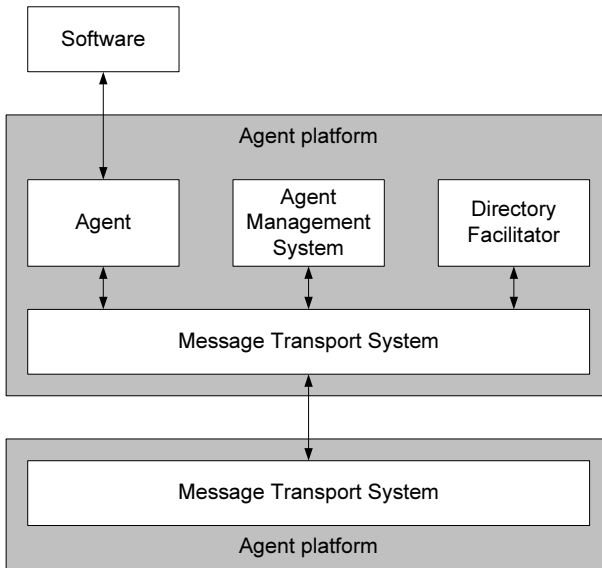


Fig. 2.4. FIPA agent management reference model constitution

FIPA Agent Message Transport Service

The FIPA Agent Message Transport Service specification (SC00067F), as part of the FIPA Agent Management specification, supports the message transportation between the interoperating agents. Two major specifications are involved, *i.e.* a reference model for an agent Message Transport Service (MTS) and the definitions for the expression of message transport information to an agent MTS [50].

A three-layered reference model is provided by MTS, *i.e.* the Message Transport Protocol (MTP) for physical messages transfer between two Agent Communication Channels (ACCs), the MTS which provides the FIPA ACL messages transportation between agents on the platform, and the ACL representations from both MTS and MTP. Figure 2.5 shows the FIPA message transport reference model. Additionally, other distinct components are involved in an agent MTS:

- two transport protocols, for transporting messages between agents using the Internet Inter-Orb Protocol (IIOP) and Hypertext Transfer Protocol (HTTP), specified by FIPA Message Transport Protocol for IIOP (SC00075G) and FIPA Message Transport Protocol for HTTP (SC00084F) respectively;

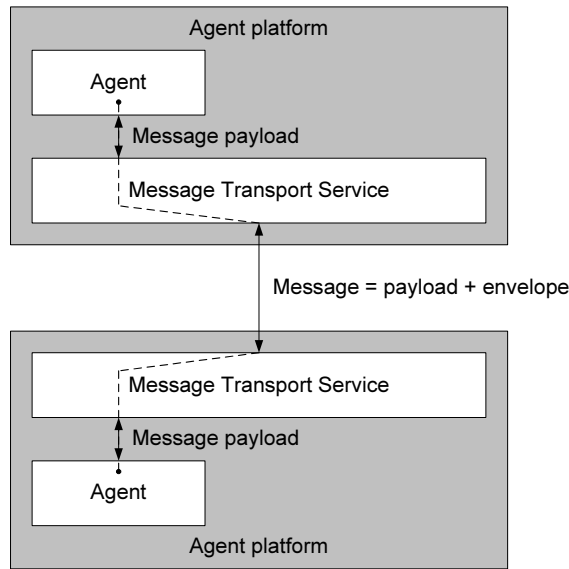


Fig. 2.5. FIPA message transport reference model

- two message transport envelope specifications, *i.e.* FIPA Agent Message Transport Envelope Representation in Extensible Markup Language (XML) Specification (SC00085J) and FIPA Agent Message Transport Envelope Representation in Bit-efficient Encoding Specification (SC00088D) which provide syntactic representations of a message envelope in XML form and bit-efficient form, respectively;
- three message representation specifications, *i.e.* FIPA ACL Message Representation in Bit-efficient Encoding Specification (SC00069G), String Specification (SC00070I) and XML Specification (SC00071E) for representing ACL syntax in a bit-efficient form, string form and XML form, respectively.

FIPA Agent Communication Standards

One of the most important areas that FIPA standardised is agent communication, which is the core category at the heart of the FIPA multi-agent system model. Four components are involved in FIPA Agent Communication specifications, *i.e.* Agent Communication Language (ACL) Message, Interaction Protocols (IPs) of message exchange, speech act-based Communicative Acts (CAs) and Content Language (CL) representations.

- FIPA ACL Message Structure specification (SC00061G) standardises the form of a FIPA-compliant ACL message structure to ensure interoperability.

- A number of different interaction message exchange protocols are dealt by FIPA Interaction Protocols (IPs) specifications, such as request and query interaction protocols, brokering and recruiting interaction protocols, subscribe and propose interaction protocols, *etc.*
- FIPA Communicative Act Library (CAL) specification (SC00037J) defines the structure of the CAL and the formal basis of FIPA ACL semantics.
- A set of languages used in FIPA Messages are denoted by FIPA Content Languages (CLs). For example,
 - a concrete syntax for the FIPA Semantic Language (FIPA SL) is defined by FIPA SL Content Language specification (SC00008I) for use in conjunction with the FIPA ACL;
 - FIPA Constraint Choice Language (CCL) Content Language specification (XC00009B) allows agent communication to involve exchanges about multiple interrelated choice;
 - FIPA Knowledge Interchange Format (KIF) Content Language specification (XC00010C) expresses the objects and propositions as terms and sentences, respectively;
 - FIPA Resource Description Framework (RDF) Content Language specification (XC00011B) constructs components of FIPA SL in the resource description framework representation.

2.4.2 Mobile Agent Standards

The mobile agent technology and its implementation are retarded due to the lack of world-wide, accepted standards which cover the most key areas of mobile agents, such as, code, data relocation, communication, interoperability, infrastructure and security. Currently, there are two main mobile agent standards available, OMG MASIF and FIPA Mobile Agent Standard.

OMG MASIF

The Mobile Agent System Interoperability Facility (MASIF) [51] is a standard for mobile agent systems, which was developed by Object Management Group (OMG) in 1998. OMG MASIF aims in migrating mobile agents between agent systems with same profile (*e.g.* language, agent system type, authentication type and serialisation methodologies) based on standardised CORBA Interface Definition Language (IDL) interfaces. The conventional client–server paradigm and the mobile technology can be integrated seamlessly based on the MASIF standard, which is built on the top level of CORBA.

MASIF is capable of interoperability between the agent systems developed in the same programming language which standardises the following concepts:

- Agent Management. As a system administrator, it provides agent systems management for different types by standard operations. Therefore, the agents can be remotely created, started, suspended, resumed, and terminated.

- Agent Transfer. Agent applications can freely move between the agent systems of different types. The agent class files are received and fetched between different platforms by agent transport methods, and available for agent management functions and execution subsequently.
- Agent and Agent System Names. The syntax and semantics of the agent and agent system names can be standardised by MATIS, which allows them to identify each other.
- Agent System Type and Location Syntax. If the agent system type is not suitable for the agent, the agent transfer will not happen. In order to locate each other, the location syntax is standardised by MATIS. In a mobile agent environment, agent tracking addresses the agents location, such as remotely query agent directories on different platforms.

However, the local agent operations such as agent interpretation, serialisation and execution are not be standardised by MASIF. In order to address interoperability concerns, the interfaces have been defined and addressed at the agent system rather than at the agent level. Two sets of interfaces constitute MASIF, such as, MAFAgentSystem and MAFFinder. So that, the specified remote operations are permitted to carry out transparently.

FIPA Mobile Agents Standard

FIPA standardised agent mobility in the FIPA Agent Management Support for Mobility specification (OC00005A) in 1998. However, this specification was not developed after 1999.

Two types of mobility are concerned in the FIPA 98 specification, mobility in devices and software. A series terms related to mobility are defined, such as the notions of mobile agents, migration, stationary agents, *etc.* In this specification, agents are allowed to take advantage of mobility by specifying the minimum requirements and technologies. Furthermore, combining with other FIPA specifications, a wrapping mechanism for existing mobile agent systems is proposed to promote interoperability.

Along with FIPA being officially accepted by IEEE Computer Society, a new working group was appointed in September 2005 to resume work on standards for mobile agents. The primary objective of this group is to improve and extend the existing specifications for agent mobility. The latest research results and experiences acquired from existing implementations and evaluation will be involved into the new specifications. Moreover, the number of reference implementations of protocols in software components will be developed as agent toolkits. The new specifications will be defined for communication interfaces and network protocols. According to the agenda, the Standard IEEE FIPA Mobile Agent specification will be reported in December 2006.

2.5 Mobile Agent Technology

Mobile agent systems are systems which involve the transfer of a currently executing program, known as a *mobile agent*, from one location to another. Fuggetta, Vigna and Picco [52] state that “in the mobile agent paradigm a whole computational component is moved to a remote site, along with its state, the code it needs, and some resources required to perform the task.” Mobile agents were first discussed in the early 1990s, and applications to a wide range of areas have been proposed or implemented. For example, [53] describes a distributed calendar application implemented using mobile agents, [54] describes a military information retrieval application and [55] describes the application of mobile agents to network monitoring.

There are a number of reasons why mobile agents might be used in any particular application:

- Mobile agents can provide performance improvements by reducing network load [54, 56].
- Using mobile agents can allow servers to be made more flexible, with components being added and removed at runtime [57].
- Mobile agents permit disconnected operation, in which a client can “launch” a mobile agent into the network, disconnect, and then reconnect to retrieve the results of the mobile agent’s task [58].

However, because of concerns regarding mobile agent security and a lack of motivation to deploy mobile agents, there has so far been limited use of mobile agents in real applications [59]. The mobile agent security issue consists of two problems: protecting a host and its data from a malicious agent or other attacker, and protecting an agent and its data from a malicious host or another agent [60]. While the first of these problems may be, at least partly, solved, there is still ongoing research into the second [61].

2.6 Multi-agent System

2.6.1 Architectures

Several architectural styles have been used in the development of multi-agent systems. Shehory [62] describes four such organisations:

- Hierarchical multi-agent systems, in which agents communicate according to a hierarchical structure, such as a tree. A system such as the Open Agent Architecture [63], which uses brokers, is a hierarchical system, as each agent communicates only with a broker or facilitator agent. Shehory gives the disadvantage of such a system as the reduction in autonomy of the individual agents, as lower levels of the hierarchy depend on and may be controlled by higher levels. However, hierarchical architectures can greatly

reduce the amount of communications required, and also the complexity and reasoning capabilities needed in the individual agents.

- Flat multi-agent systems, in which any agent may contact any of the others. These provide the greatest agent autonomy, but result in more communications between agents. Also, agents in a flat structure must either know the locations of their communications partners, or be provided with agent location mechanisms such as yellow pages services. Many smaller multi-agent systems appear to be developed using a flat organisation.
- A subsumption multi-agent system is a system in which agents are themselves made up of other agents. In this system, the subsumed agents are completely controlled by the containing agents. This is similar to the subsumption architecture for an individual agent. According to Shehory, the fixed structure of a subsumption multi-agent system provides efficiency but restricts the flexibility of the system.
- A modular multi-agent system is comprised of a number of modules. Each module normally employs a flat structure, while intermodule communications is relatively limited. A modular multi-agent system might be useful in a situation such as power system automation, in which each substation could be categorised as a single module. Most communications within a power system are either within a substation or between a substation and the control centre, and so this might be an appropriate multi-agent system structure.

2.6.2 Multi-agent Programming

Logic-based Agent Programming Language

Jason and the Golden Fleece of Agent-oriented Programming

Jason [64] was designed as an interpreter language which is an extended version of AgentSpeak(L) [65]. As a pure logic-based agent programming language, *Jason* is based on the BDI architecture and BDI logics [32]. *Jason* allows agents to be distributed over the network through the use of SACI [66]. It implements the operational semantics of AgentSpeak(L) and the extension of that operational semantics to account for speech-act-based communication among AgentSpeak agents as given in [67] and [68], respectively.

Besides interpreting the original AgentSpeak(L) language, some of the other features available in *Jason* are given below [69]:

- speech-act based interagent communication (and belief annotations on information sources);
- annotations on plan labels, which can be used by elaborate selection functions;
- the possibility to run a multi-agent system distributed over a network (using SACI);

- fully customisable (in Java) selection functions, trust functions, and overall agent architecture (perception, belief-revision, interagent communication, and acting);
- straightforward extensibility by user-defined internal actions, which are programmed in Java;
- clear notion of multi-agent environments, which can be implemented in Java.

Cognitive Agents Goal Directed 3APL

Another multi-agent programming language worthy of mention is Goal Directed 3APL [70]. This language is designed to separate the mental attributes (data structures) and the reasoning process (programming instructions), which is motivated by cognitive agent architectures. One of the main features of 3APL is that it provides programming constructs to implement mental attitudes of individual agents directly, such as beliefs, goals, plans, actions, and practical reasoning rules. The basic building blocks of plans are composed of actions, which fall into different categories including mental actions, external actions and communication actions. An agent's belief base as well as the shared environment can be updated and modified in the implementation of selection and execution of actions and plans, which are allowed in the deliberation constructions.

An individual 3APL agent can be implemented by the 3APL programming language. Moreover, the shared environment, which the agent can be performed on, is designed by Java programming language as a Java class. Besides interacting with the environment, a 3APL agent can contact with each other agent via direct communication. Additionally, a 3APL multi-agent system is composed of a number of concurrently executed agents. The 3APL platform is built for sharing an external environment of a group of agents. Therefore, the platform allows the implementation and parallel execution of a set of agents and achieves the function of the 3APL agent programming language.

Using 3APL, the agents can be implemented of communicating with each other, observing the shared environment, reasoning about and updating the states and executing actions in the shared environment.

Java-based Agent Programming Language

JADE - A Java Agent Development Framework

JADE (Java Agent DEvelopment Framework) is a Java framework for the development of distributed multi-agent applications [71]. The JADE platform is open source software delivered by TILAB (Telecom Italia LABORatories).

JADE can be seen as an agent middleware which provides a number of available and convenient services, such as DF which provides a yellow page

service to the agent platform. In addition, server graphical tools for debugging and testing the agent programs are also supported by JADE. One of the main characters of the JADE platform is that it strictly adheres to the IEEE computer science standard FIPA specifications to provide interoperability not only in the platform architectures design, but also in the communication infrastructures. Furthermore, JADE is very flexible and compatible with many devices with limited resources, such as the PDAs and mobile phones.

The goal of JADE is to simplify development while ensuring standard compliance through a comprehensive set of system services and agents [71]. To achieve such a goal, JADE offers the following features to the agent programmer:

- FIPA-compliant Agent Platform. Three agents are automatically activated with the JADE platform start-up, which includes the Agent Management System (AMS), the DF, and the Agent Communication Channel (ACC);
- distributed agent platform. The agent platform can be split on different hosts. Each host just executes only one Java application. Therefore, parallel tasks are processed by one agent and scheduled in a more efficient way by JADE;
- rapid transport of ACL messages inside the same agent platform;
- graphical user interface for organising different agent and agent platforms.

Jadex: A BDI Reasoning Engine

Jadex [72] is a software framework conducted by the Distributed Systems and Information Systems Group at the University of Hamburg. Following the BDI model, Jadex focuses on creation of goal-oriented agents. The framework supports the development of a rational agent layer on top of the middleware agent infrastructure, such as the FIPA compliant JADE platform. Therefore, the extended BDI-style reasoning and FIPA compliant communication are combined together in Jadex.

As a BDI reasoning engine, Jadex addresses the conventional limitations of BDI systems. Four basic concepts are integrated into Jadex, which are beliefs, goals, plans and capabilities. The reaction and deliberation mechanism in Jadex is the only global component of an agent [69]. Four goals are supported by Jadex, such as, a perform goal directly corresponding to the execution of actions, an achieve goal in the traditional sense, a query goal regarding the availability of agents required information, and a maintain goal for tracking a desired state.

After initialisation, based on internal events and messages from other agents, the agent is executed by the Jadex runtime engine by tracking its goals. In addition, many pre-defined functionalities and third party tools are integrated into Jadex, such as the ontology design tool Protégé [73].

JACK Intelligent AgentsTM

JACK Intelligent AgentsTM [74] was designed based on the BDI model and supported an open environment for developers to create new reasoning models. The JACK Intelligent AgentsTM framework was developed by a company called Agent Oriented Software, which brings the concept of intelligent agents into the mainstream of commercial software engineering and Java [75]. JACK Agent Language (JAL) is based on Java, which provides a number of light-weight components with high performance and strong data typing.

In addition, JAL extends Java in constructions in three main aspects to allow programmers to develop the components, such as: [75]

- a set of syntactical additions to its host language;
- a compiler for converting the syntactic additions into pure Java classes and statements which can be loaded with and be called by other Java code;
- a set of classes called the kernel provides the required runtime support to the generated code.

2.6.3 Middle Agents: Brokers and Facilitators

In a system consisting of only a few agents, it is possible to hard-code knowledge about other agents in the system, or to use a broadcast-based protocol, such as the Contract Net Protocol [76], in order for agents to locate others that are capable and willing to co-operate with them. However, as the number of agents increases, it becomes more and more difficult for agents to locate others which provide services that they require. This problem can be addressed by the use of middle agents, of which there are two main varieties: *matchmakers* (or *facilitators*) and *brokers*. A *matchmaking* system uses a set of agents that can be used by others to register their services. Other agents use these matchmakers to find agents providing a service that they require. Once they have done so, they communicate directly with that agent. A simple example of a matchmaking system is the FIPA Directory Facilitator [77].

In a *brokered* system, there are a number of agents, known as brokers. As with a matchmaker, agents can register their services with a broker. However, when a client wishes to use a broker, instead of searching for a suitable service-providing agent, they submit their request to the broker, which then handles the locating of a suitable agent, and sends the request to that agent. The client agent never communicates directly with the service-providing agent.

Decker, Sycara and Williamson [78] analyse the relative advantages and disadvantages of the brokering and matchmaking approaches. Their results suggest that the amount of time elapsed between a request being made and the same request being fulfilled is lower in a brokered system than in a match-making system, and that a brokered system can respond more quickly to the failure of service providers. However, they also state that in a brokered system, the broker is a single point of failure, and therefore the whole system

can fail if a broker does. They conclude that a hybrid system containing both matchmakers and brokers may permit the advantages of both methodologies to be realised.

Kumar, Cohen and Levesque [79] use a set of brokers which work together as a team to prevent problems which arise as a result of broker failures. Each broker registers its service-providing agents with the team as a whole. If a broker fails, the other agents in the team will attempt to connect to the agents that were served by that broker. Once an agent has been reconnected in this manner, the broker that did so announces this fact to the other brokers, which will then not attempt to connect to that agent. The broker team is also responsible for maintaining a set number of brokers in the system.

2.7 Agent Application Architectures

There has been much previous work in the field of application architectures for multi-agent systems. For example, the RETSINA [80] architecture is a 3-tier architecture consisting of user agents, wrapper agents representing information sources and “middle agents” which transfer data between the two. Wrapper agents both “agentify” the data sources, allowing them to be queried using the agent communication language and convert data from the data models (ontologies) used by the individual data sources into a global ontology used for querying. Therefore, the wrappers assist in integrating data from heterogeneous data sources. RETSINA has been applied to several problems including financial portfolio management and visit scheduling. A similar information architecture, consisting of an ontology agent and database agents, is described in [81]. These architectures provide a basis for the development of a multi-agent information management architecture. However, they do not include the other functions used in power system automation, such as information management and control. Therefore, it is necessary to significantly extend these architectures to include this functionality.

The InfoMasterTM [82] system used the Knowledge Query and Manipulation Language (KQML) [83] to provide a uniform interface to a number of heterogeneous data sources. The architecture of infomasterTM consisted of data sources, each having their own wrapper, the “InfomasterTM Facilitator” which performed data integration, and various user interfaces such as a Web interface. InfomasterTM used rules to translate from individual database schemas into a global schema. Similar techniques are used by many information integration systems, including the multi-agent information management system described in this book.

A number of multi-agent systems have been employed to handle various aspects of industrial automation. The ARCHONTM project [84] developed an agent architecture which was used in a number of applications. ARCHONTM agents consisted of two layers: the *ARCHONTM layer*, which was responsible for local control, decision making, agent communications and

agent modelling, and the *AL-IS interface*, which handled communications between the ARCHONTM layer and the “intelligent system” being wrapped by the agent .

There are many applications of agent technology in the manufacturing industries. These are similar in some ways to applications in the process industries and utilities, but often focus on machine control and task allocation. For example, the PABADIS project [85] aims to develop a system for agent-based manufacturing. The PABADIS system contains agents representing machines and products [86]. The machine agents register descriptions of their capabilities with a lookup service, which can be used by the mobile product agents to locate machines capable of carrying out the tasks involved in manufacturing a particular product. The PABADIS architecture demonstrates the use of a multi-agent system in an industrial process. However, the architecture of a manufacturing system differs from that of a utility. A utility system is a continuous process, whereas a manufacturing system contains discrete parts and outputs. Also, the area covered by a distributed utility system is much wider than a single factory. In this work it is hoped to make use of the general agent-oriented principles used by PABADIS (use of directories and representation of components of the plant as agents) but to design an architecture more suited to the power systems and continuous process field.

Bussmann and Schild [87] used a multi-agent system in the control of a flexible manufacturing system. The system was used to manage the flow of material between different machines, and to allocate tasks to machines. It was applied to automobile manufacturing. The approach taken by this system was based on auctions, in which workpieces auctioned off tasks to machines. It was found that this system provided both improved throughput and increased robustness compared to traditional methods [88]. As with PABADIS, the relevance of Bussmann and Schild’s work is restricted by the fact that their application is in the manufacturing domain.

Leito and Restivo [89] describe a multi-agent architecture under development for “agile and cooperative” manufacturing systems. As well as controlling the manufacturing system, the architecture supports re-engineering of products. The agent architecture consists of Operational, Supervisor, Product, Task and Interface agents. This architecture is also intended for use in manufacturing industries. However, it might be possible to use agents corresponding to the operational agent (which Leito and Restivo define as corresponding to the “physical resources”) and supervisory agent in a utility system. Also, as the paper states, “only preliminary results are presented”, and further work is therefore required.

Mangina *et al.* have also developed the condition monitoring multi-agent system (COMMAS) architecture [90], which uses three layers of agent. Attribute Reasoning Agents (ARAs) monitor and interpret sensor data, Cross Sensor Corroboration Agents (CSCAs) combine data from different sensors, and Meta Knowledge Reasoning Agents (MKRAs) provide diagnostics based on the information provided by the other agents. Mangina’s architecture is

relevant to the power systems domain. However, it provides only the single task of condition monitoring, and does not contain information management and remote control or operation functionality.

These applications have been relatively successful, suggesting that the multi-agent approach is a promising method for the implementation of industrial automation systems. However, the previous work described does not provide a single architecture for providing all the functions required by a power system automation system, either because it focuses on a single application or because it is intended for use in manufacturing industries. The work described in this book is intended to provide such an architecture.

IP Network-based Multi-agent Systems for Industrial
Automation

Information Management, Condition Monitoring and
Control of Power Systems

Buse, D.P.; Wu, Q.H.

2007, XVII, 187 p., Hardcover

ISBN: 978-1-84628-646-9