Multilingual Knowledge Based European Electronic Marketplace

Technology Review and Selection
Public Release

France Telecom - R&D

Sema Group sae

Universidad Politecnica de Madrid - UPM

National Technical University of Athens - NTUA

Centre National de la Recherche Scientifique - CNRS

Tradezone International Ltd - TZI

Technical Research Centre of Finland - VTT

Ellos Postimyynti Oy

Société Nationale des Chemins de Fer Français - SNCF

Fiduciaire Juridique et Fiscal de France - FIDAL

Project Reference N° IST-1999-10589

Project ref. no.	*IST-1999-10589*
Project acronym	**MKBEEM**
Project full title	**Multilingual Knowledge Based European Electronic Marketplace**

Security (distribution level)	*Public*
Contractual date of delivery	*M03: May 2000*
Actual date of delivery	*Wednesday, 31/05/2000*
Deliverable number	*D51*
Deliverable name	*Technology Review and Selection*
Type	*Report*
Status & version	*Final (v1.1)*
Number of pages	*1*
WP contributing to the deliverable	*WP5*
WP / Task responsible	*NTUA*
Other contributors	*VTT, UPM, FT R&D*
Author(s)	*NTUA: Yiannis Kouroupis, Yiannis Stavroulas, Katerina Tsiara, Theodora Varvarigou* *VTT: Aarno Lehtola, Kuldar Taveter* *UPM: Víctor A. Villagrá, Jorge E. López de Vergara* *FT R&D: Christophe Duhem*
EC Project Officer	*Yves Paternoster*
Keywords	*MKBEEM, agent technology, agent platforms, rational agent, FIPA, ACL, KQML*
Abstract (for dissemination)	*The MKBEEM project aims at the development of a distributed system, providing intelligent multilingual mediation services to e-commerce platforms. This report aims to anticipate the needs of the system in terms of communication between its various modules, and review the methodologies and tools that can be used to that end. It comprises a preliminary study of the technological needs of the MKBEEM project with respect to process communication, and a survey of existing technologies that could be employed.* *In particular, the report focuses on emerging agent technology methodologies and tools, assessing their usability in the frame of the project. It provides an insight in the potential of agent communication methodologies and a critical review of off-the-shelf agent platforms that implement such methodologies.*

Version history

Version	Date	Comments and Actions	Status
0.2	04/04/2000	Contribution from NTUA, VTT	Draft
0.4	08/05/2000	Additions from NTUA, VTT and UPM	Draft
0.5	20/05/2000	Additions from NTUA and FT R&D	Pre-final
0.97	26/05/2000	Final additions and corrections	Pre-final
1.00	29/05/2000	Updating of section 3.6.1.3, correction of typos	Final
1.01	30/05/2000	Minor rephrasing, removal of Annex A (ZEUS prototype)	Final
1.1	25/01/2001	Minor corrections, replacement of logo	Public release

TABLE OF CONTENTS

Executive summary

The MKBEEM project aims at the development of a distributed system, providing intelligent multilingual mediation services to e-commerce platforms. The D51 document gathers and reviews information about the latest and most popular methodologies, platforms and tools in agent technology, compares them and comments on their suitability for the MKBEEM project. Its primary focus is the communication and management needs of the distributed modules of the MKBEEM system. A large part of the document is dedicated to technology following the agent paradigm, which is an attractive candidate methodology for modelling the system.

Factors that affect the suitability of technologies for the project are discussed first. Such factors can range from the abstract down to the practical level. Concerning the methodologies, the most important factors are the degree in which they are standardised, the possibility of their prevalence over similar methodologies and the wealth of agent characteristics they can support. Furthermore, the report concludes that individual tools should be judged mainly on the basis of the methodology they implement and the practical aspects, such as efficiency, reliability, ease of use etc. These are the decisive factors that will eventually determine, along with the user requirements specification, whether such an off-the-shelf platform could be used or a custom one should be built.

The most important standardised methodologies of agent technology are the FIPA specifications, KQML by the DARPA Knowledge Sharing Effort and MASIF by OMG. Of those, the FIPA specifications appear to be the best choice for the current project. The existence of high-quality implementations of these methodologies is an important factor taken into account to reach this conclusion.

There are a number of FIPA-compliant agent development platforms available, the most important ones being ZEUS, JADE and FIPA OS. These are reviewed, along with JKQML, JATLite and Grasshopper. The former two follow the KQML paradigm, while Grasshopper is a MASIF implementation. The final conclusions from their comparison are set forth in the last chapter of this document.

Technology Review and Selection

1 Introduction

This report is the first deliverable in Work package 5 of the MKBEEM project. WP5 contains the tasks related to the realisation of the Rational Agent. The Rational Agent will be the core of the system, linking the other modules. The Rational Agent will contain the search engine that will process the users' queries. Furthermore, according to our understanding of the system's global architecture so far, the Rational Agent will be responsible for the co-ordination of MKBEEM's various modules. It is a part of the work in WP5 to provide feedback on the available methods for the communication of these modules.

The goal of this report is thus to review technologies that can be used for the communication among the distributed modules of the system, compare them in the context of MKBEEM and offer an insight on their suitability for the project at hand. It focuses on the emerging agent standards and specifications, and off-the-shelf products that have resulted from them, with the intention to determine whether some of them can be used to facilitate the system's development.

The report is not meant to be conclusive towards the selection of technologies and tools, as this decision depends greatly on the final specification of the user requirements (project deliverable D21). It is meant, on the other hand, to highlight the features and capabilities of various communication technologies, so that they can be effectively matched against the user requirements when they are available, thus guaranteeing a rational choice.

1.1 Document Structure

The report is structured in four chapters, the first being this introduction. The second chapter sums up an analysis of the aspects that must be considered for the evaluation of technologies and tools. In chapter three technologies and tools are reviewed. Chapter four comprises the conclusions of the review.

1.2 Glossary & Acronyms

ABROSE: Agent Based Brokerage Services in Electronic Commerce

ABS: Architecture for a Brokerage Information Service

ACC: Agent Communication Channel

ACID: Atomicity, Consistency, Isolation, and Durability

ACL: Agent Communication Language

AD: Agent Domain

ADS: Agent Domain Service

AEW: Agent-enhanced Workflow

AMR: Agent Message Router

AMS: Agent Management System

AOIS: Agent-Oriented Information Systems

AP: Agent Platform

APL: Applicable Plan List

ATP: agent transfer protocol

BA: Belief Agents

BDI: Belief-Desire-Intention

BN: Belief Network

BT: British Telecom

CN: Conceptual Network

CORBA: Common Object Request Brokerage

DF: Directory Facilitator

FIPA: Foundation for Intelligent Physical Agents

GUI: Graphical User Interface

GUID: Globally Unique Identifier

HAD: Heterogeneous, Autonomous and Distributed systems

IIER: Intelligent Information Exploitation and Retrieval

IIOP: Internet Inter-Orb Protocol

IPMT: Internal Platform Message Transport

JADE: Java Agent Development kit

JAFMAS: Java-based Agent Framework for Multi-Agent Systems

JAT: Java Agent Template

JESS: Java Expert System Shell

JIDM: Joint InterDomain Management

JKQML: Java-based KQML

JVM: Java Virtual Machine

KAPI: KQML API

KIF: Knowledge Interchange Format

KQML: Knowledge Query and Manipulation Language

KSE: Knowledge Sharing Effort

KTP: KQML Transfer Protocol

MAS: Multi-Agent System

MIB: Management Information Base

OTP: Object Transfer Protocol

PAF: Personal Agent Framework

RA: Rational Agent

RMI: Remote Method Invocation

RPC: Remote Procedure Call

SL: Semantic Language

SMART: Stationary and Mobile Agent Resource Toolkit

SMTP: Simple Mail Transfer Protocol

SQL: Structured Query Language

SSL: Secure Socket Layer

TCP/IP: Transmission Control Protocol / Internet Protocol

TIIERA: Tactical Intelligent Information Exploitation and Retrieval Agents

UDP: User Datagram Protocol

UML: Unified Modelling Language

WAP: Wireless Application Protocol

XML: Extensible Mark-up Language

2 Important Selection Factors and Considerations

The MKBEEM project is committed to the use of standardised technologies and off-the-shelf tools as much as possible. The selection of technologies and tools in an emerging field is not simple, and several factors have to be taken into account:

- *Standardisation*. Typically, standards are meant to provide for re-usability and inter-operability. In the case of MKBEEM this is a very tender issue, because, as we point out later, inter-operability is very important in agent technology. However, it is still an emerging technology and no formal standards, e.g. by international, commonly accepted standardisation bodies, exist to date.
- *Future*. Currently there is a number of different agent technologies, evolving in parallel. It is quite clear that not all of them have a future, but it is not so clear which ones do. Care must be taken to select technologies that appear to be trendy and promising.
- *Agent Characteristics*. Intelligent agents do not always demonstrate the same amount of intelligence. It is important to decide what agent characteristics are desirable for the project at hand. Thus we can avoid approaches that fall short of our expectations and others that are too broad, especially since usually there is a trade-off between broadness and efficiency.
- *Practical Aspects*, such as the quality and availability of existing tools, or the ease of use, also must be taken into account.

In the next sections, we give a detailed account of the agent characteristics and their importance in the scope of our project, and of practical issues that also affect our choices.

2.1 Agent Characteristics

In the Intelligent Agent literature, one can find a variety of properties and capabilities attributed to agents. In this section we review the fundamental ones and consider their necessity in the frame of the MKBEEM project, as well as their potential weight as technology selection factors.

2.1.1 Autonomy

Autonomy is an issue, which directly or indirectly causes many of the problems faced when developing and using systems that cross organisational borders while operating. These HAD systems (Heterogeneous, Autonomous and Distributed systems) usually implement inter-organisational processes and one can find such systems e.g. in the area of e-commerce. The reason for the problems is simple, the processes cross the borders of autonomous organisations or relatively autonomous organisational units. Intuitively, organisational (O-) autonomy means that the organisation cannot be controlled by another through (some) interactions. The organisation has thus self-determination in this respect.

O-autonomy is the source of other forms of autonomy, which are of interest here, like design (D-), management (M-), communication (C-), and execution (E-) autonomy. A D-autonomous organisation is able to determine for itself how its computer hardware and software architecture, telecommunication infrastructure etc. is composed. This leads easily to heterogeneity between the technical infrastructures of different organisations and even organisational units. M-autonomy means that an organisation can determine for itself how the systems are used (policies, security, level of service, etc.). One of the main consequences of M-autonomy is that computers (e.g. servers, but especially clients) and other terminal systems (e.g. mobile phones) can be unreachable through a network. Thus, they behave in a C-autonomous manner. Another consequence of M-autonomy is E-autonomy, which means that a computer does not need to perform at all measures indicated in a request or can execute the measures at its own pace and in the way best suited for the organisation. As discussed in [24] a solution to the problems incurred by heterogeneity and autonomy is to establish a homogeneous global domain.

Assume that a remote call is requested for a business offer. The parties must have agreed on e.g. how the object of the offer is to be specified in common terms (part of the common process model). Design and execution autonomy means that the parties can hide how they internally process the offers. Design autonomy lets them implement the semantics of the operations in a manner more appropriate for them. Execution autonomy allows them to decide for instance whether they execute the message at all or how long it will take. One reason for refusing to execute a message may be failure in authentication

of the calling remote system. Communication autonomy means that a remote system can be unreachable through the network, basically as long as it wills.

In MKBEEM case the agent framework communicates with the content/service providers via their interface agents, with e-commerce platforms and with end-users. The agent framework constitutes a kind of "systems glue" mediating the processes and information between the parties. Within itself it constitutes a homogeneous platform but it has to connect with heterogeneous systems of content/service suppliers and e-commerce operators. A good agent framework would include necessary services for tracking and controlling the flow of the overall process. This would involve, for instance, tracking timing constraints and automatically starting compensating or contingency activities when a C-autonomous party is unavailable or has failed. Support for asynchronous communication would be necessary, as far as E-autonomy leaves response times arbitrary and the overall transactions can be long-living. If these properties are not covered by the infrastructure of the chosen agent platform, they must be possible to easily program on the application level.

With agent frameworks, *autonomy* is often defined as the ability of an agent to act without external help. This refers not so much to the sufficiency of the agent's own means and resources, but rather to its ability to adapt itself to the task at hand without external intervention. Autonomy in this sense is an absolute requirement for the Rational Agent, since it will have to plan its own actions and essentially judge its success in the completion of its tasks with little, and implicit at that, user guidance. However, we do not expect it to be an important issue concerning the technology selection, since autonomy in that sense is a matter of internal design of the RA. The only prerequisite from the underlying technology is that it should allow the RA to initiate actions (i.e. ask questions to the user), rather than merely responding to actions initiated by others.

2.1.2 Reactivity, Sociality

Reactivity is the ability to perceive the state of one's environment, and act according to it. It usually entails an explicit model of the environment. *Sociality* is the ability to communicate with other agents, in order to achieve one's goals. Sociality goes far beyond classical process communication, because it implies that an agent should be able to communicate with others in a dynamic, rather than a pre-defined way. This means that a social agent should be able to establish communication and co-operate with another agent, if that serves its purposes, *even if such co-operation or the existence of the other agent was not foreseen during its design*. To achieve this, agents must possess a commonly understood communication language, in which service requests and replies could be specified, as well as assertions, claims and other actions necessary for the exchange of knowledge.

Similarly, reactivity and sociality are also necessary properties of the RA. While reactivity does not pose strict requirements on the underlying technology, sociality requires the use of an agent communication language. It is, therefore, important to give an insight on why we consider the social ability of MKBEEM agents important.

Agent technology has matured significantly along the past five years. A key property of multi-agent systems is *openness*, a property that the agent community is still striving to define and guarantee [18]. The current popular understanding is that an open agent should be able to accept and integrate new agents, possibly offering novel services, in the course of its evolution. Some traditional software systems claim to have an open architecture, but this claim is poorly justified, if we compare it with the relevant claim of agent systems. The farthest that we can stretch the openness of traditional software systems, is to say that new modules can be integrated given that they use a certain, *system-specific* interface.

Agent systems are much more ambitious: the current trend is towards a universal ACL, common for all the agents throughout the world. Were this vision to materialise, we would have an environment where truly innovative, intelligent applications could be implemented. This is one of the aspects of agent systems that distinctively distinguishes them from typical distributed computing systems.

Specifically for the project at hand, we can immediately point out that there is no logical restriction why there should be, for example, only one RA in the system. The openness of the system could allow other agents to offer similar services. Similarly, the RA could co-operate seamlessly, without the need for customisation, with other user agents, e.g. ones designed for WAP interfaces, or agents providing directory services. Thus, the trade-off between the social ability of the MKBEEM agents and the restrictions this poses on the underlying technology seems to be in favour of the former.

2.1.3 Mobility

Mobility is the capability of an agent to *migrate* to another agent platform, when this is considered an advantage with a view to its current goals. Mobility is one of the most promising features of agent technology and there is a vast number of potential applications that draw on it, primarily related to distributed processing and distributed databases.

A typical example of this concept is the mobile Internet search agent. This is imagined as an agent that is created on the desktop of a user that wants to perform a search on the Internet. First, the user loads search parameters to the agent. Then the mobile agent travels through the Internet, “docking” on several platforms offering search services. On each platform, the agent can make use of the local computational resources and information, without delays or bandwidth limitations. Finally, when the agent is content with the information found so far, it returns to its home platform and presents the information to the user. Of course, this example assumes that the search agent also possesses very high degrees of autonomy and reactivity.

Despite its promise, agent mobility currently has little to offer to MKBEEM. The reason for this is that although there is a degree of distribution of the processes and databases involved, all of the agents will be running on the same platform. Note that an agent platform is not necessarily confined to a single computer, or even a LAN. Hence it is perfectly possible, for example, for the User Agent to reside on the user’s PC and the RA at the project marketplace’s back-office, while still being on the same platform.

However, we can foresee a use for mobile agents in a subsequent project, where there would be a number of different MKBEEM marketplaces, each encompassing a group of CP/SPs. Then it would make sense for the RA to be able to travel and dock to a remote marketplace, in order to perform its queries locally. Consider that answering a user’s query might require several iterations of the refine-query-process results sequence. Consequently, it would be a good choice to base MKBEEM on a technology that supports agent mobility, if only for the sake of expandability.

2.1.4 Security Considerations

The MKBEEM system, the heart of which is the RA, mediates in a trading process between the end-user and the CS/SP services and e-commerce platform. It is important that the agent platform within itself contains proper authentication and data protection so that no false agents can get hooked into the trading processes. This relates to the way mutual trust dependencies are handled in the agent community.

As far as the MKBEEM system mediates information that is part of the legal contract between the user and the seller the system has to authenticate reliably all the parties it is communicating with, both the CP/SPs and e-commerce operators and the end-users/clients. In international commerce, it is important to authenticate also the location of the client as it reflects to the conditions on which the goods or services are provided. The messaging between the parties must be encrypted whenever confidential information is transferred, like account information. The data security services could be part of the agent platform. The minimum requirement is that they can be practically implemented.

2.1.5 Transaction Support

Transaction support in MKBEEM has a broader definition than with database management systems. The total process where MKBEEM system participates as an autonomous mediator constitutes a HAD system (Heterogeneous, Autonomous and Distributed system). Transaction support for MKBEEM means preserving the required transactional properties of the total process so that the supported business transactions are executed correctly. While MKBEEM system concentrates on language adaptation and use dialogue between a foreign user and CP/SP's and e-commerce services, the finalisation phases of the business transaction (e.g. money transfers) are taken care of by the e-commerce platform to which MKBEEM system has been connected. However, as an autonomous mediator the MKBEEM system is part of the transaction and must fulfil certain transactional properties itself.

Electronic trading is often characterised by long-living transactions, high security requirements, multiphase protocols, involvement of many actors in the process, and embedding of various legacy systems in the total process [25]. Trading processes are required to fulfil transactional properties, such as atomicity guaranteeing termination of the execution in a known state. MKBEEM system participates in the transaction process, when it mediates between the end-users, the content/service providers, and the e-commerce platform. While serving in this role, it is required to support those transactional properties that are essential in the total trading process.

Traditional database transaction models are too restrictive for applications like trading between autonomous parties. The ACID properties (Atomicity, Consistency, Isolation, Durability) familiar from database transactions usually cannot be fulfilled in HAD systems. For instance, isolation is often relaxed in long-living transactions, as the data concerned cannot be locked for the whole transaction time. This stems for instance to airliners' seat reservations systems where browsing of the supply and final reservation are handled in separately causing conflicts while the browsed seats are no more available at the moment of reservation. Moreover, there are situations when rollback or compensation is not possible, at least not with zero cost. For instance, hotel reservations may require deposits making later cancellation undesirable. Ordered goods may not be cancellable after they have entered the logistics system of a third party. Several enhanced transaction modelling methodologies have been proposed, which help engineering HAD systems like electronic trading [26], [27], [28]. While mediating offers and trades the MKBEEM system must maintain audit trails that can be referred to when solving reported problems or errors. Moreover, the audit traces can in later generations of the system be used for data mining and building one-to-one marketing services. Audit trails are necessity in brokerage systems aimed for real-world use.

Some of the transaction issues outlined earlier can be tailored in the business rules of the brokerage service, like guiding the user in case of resource conflicts due to relaxed isolation (e.g. goods not available at the time of final order). This necessitates that the agent framework provides a practical knowledge representation formalism to use. Some of the transactional services should be prevalent diffusely in interfaces and provided by the platform. These include synchronising asynchronously coming messages of the external systems to the right trading processes. These also involve tracking and solving service calls that have been frozen due to communication or execution problems at the autonomous provider's site. Service calls to content provider systems may for instance (1) be refused due to communication failure or by explicit denial, (2) end-up in error state (empty query result as a special case), (3) take arbitrarily long time but in the end succeed, (4) end-up hanging. The agent frameworks could implement control primitives in their infrastructure for the cases like the ones listed. However, the minimum requirement is that the selected framework has suitable facilities for programming the execution control.

2.1.6 Manageability

Agent Platforms are currently used in many Management Systems. Many of them have been developed in well-known European Projects. However, the question here is not to use this platform for management, but to manage it.

The Management is the set of activities devoted to control and to monitor the resources of a system. Its main objective is to guarantee the service level of the managed resources with the minimal cost. Thus, the manageability of an Agent System can be an important aspect to be taken into account to improve the quality of service given to the final users.

FIPA has released an Agent Management Specification [29], in which they have defined an Agent Management System. This concept is also contained in [30]. The AMS is responsible for managing the operation of an Agent Platform (AP). These responsibilities include creation of agents, deletion of agents, deciding whether an agent can dynamically register with the AP (for example, this could be based upon agent ownership) and overseeing the migration of agents to and from the AP.

It is also very interesting to manage the Agent System not only in the way of the FIPA or OMG specification, the configuration of the AP, but also in all the functional areas of the OSI Management System [31]: Fault, Configuration, Accounting, Performance and Security. With this, an operator could be advertised of a fault in the system, perform the changes needed to correct it, obtain the usage statistics, measure the response times or change the access level of a particular user. Some of these questions are closely related to the underlying network and system resources, so that, it will be very useful to do the management of both things through the same interface, which could show the global behaviour of the whole system.

As it has been proven in [32,33], the defined management information is a key aspect for the management of a service: it decides what will be controlled or monitored and thus, what will be managed. Other important questions related to the management of a service are the management interface (how to access to such information) and the management instrumentation (how to maintain such information in the agents).

In conclusion, the manageability, including those characteristics related to integration, information, accessibility and instrumentation, is a practical aspect to be considered in the election of the Agent System.

2.1.7 Support for Business Rules

E-commerce ontologies made use of by the Rational Agent contain the rules describing legal and business issues related to E-commerce in different countries. These rules can, for example, describe the rights to sell one or another product in different countries, information about luxury taxes applicable in different countries, rules on converting prices to different local currencies, and information on methods of payment and methods and costs of delivery.

For the sake of efficiency of enforcing business rules, the representation formalism for business rules by the E-commerce ontology should be closely related to the formalism of representing knowledge for the Rational Agent. Therefore a combination of JESS Rule Language (v. section 3.2.1.1) and a JADE-based agent (v. section 3.2.1.) supporting it could be appropriate. Moreover, if we want to create a really effective and usable system, which is one of the goals of the MKBEEM project, business rules initially described by an ontology should be made operational e.g. in the form of reaction rules of "vivid agents" (v. section 2.2.1) or plans of BDI agents (v. section 2.2.2).

2.2 Agent Architectures

Agent architecture is a specific collection of software (or hardware) modules, typically designated by boxes with arrows indicating the data and control flow among the modules. A more abstract view of agent architecture is as a general methodology for designing particular modular decompositions for particular tasks [23]. Agent architectures can be thought of as software engineering models of agents. In the following, we will provide an overview of some popular agent architectures that might be relevant for the MKBEEM project.

2.2.1 "Vivid Agent" Architecture

The so-called "vivid agent" architecture described in [4]. A "vivid agent" is described to be consisting of three components:

- a *virtual knowledge base*[1] *X*, consisting of the agent's *beliefs*;
- an *event queue EQ*, i.e. a buffer receiving messages from other agents or from perception subsystems running as concurrent processes;
- a set of *reaction rules RR* determining the agent's reactive and communicative behaviour.

Agents communicate in some high-level *agent-communication language*, such as KQML [12] and ACL proposed by FIPA [5] that is based on *typed messages* such as "ASK", "TELL", "REQUEST", and "PROPOSE". In contrast to the application-specific messages in OO-programming, ACL message types are application-independent and therefore, *in combination with an ontology*, defining the semantic vocabulary of a problem domain, allow for true software interoperability [6].

Reaction rules encode the behaviour of an agent in response to perception events created by the agent's perception subsystems, and to communication events created by communication acts of other agents. Both perception and communication events are represented by incoming messages of an agent [6].

There are three types of reaction rules [6]:

- *epistemic reaction rules* of the form *Eff* ← **recvMsg**[*m(c)*, *j*], *Cond* where the event condition **recvMsg**[*m(c)*, *j*] is a test whether the event queue *EQ* of the agent contains the message *m(c)* sent by agent *j*, *Cond* refers to the agent's information state represented in its VKB, and *Eff* is an epistemic effect formula specifying a corresponding update of the agent's VKB;
- *physical reaction rules* of the form **do**(α), *Eff* ← **recvMsg**[*m(c)*, *j*], *Cond* where **do**(α) calls the procedure α affecting some actuators available to the agent;
- *communicative reaction rules* of the form **sendMsg**[*m'(c')*, *i*], *Eff* ← **recvMsg**[*m(c)*, *j*], *Cond* where **sendMsg**[*m'(c')*, *i*] is a procedure call to send the message *m'(c')* to agent *i*.

Additionally there are derivation rules of the form *Conclusion* ← *Premise,* which define intentional, predicates in the agent's virtual knowledge base [6].

[1] An agent's *virtual knowledge base* (VKB) is called "virtual" because it is not necessarily implemented as a classical knowledge base.

In the MKBEEM situation, communication events would correspond to messages received by the Rational Agent from User Agents and CP/SP Agents, and communicative actions would respectively be messages sent by the Rational Agent to User Agents and CP/SP Agents. The epistemic and communicative reaction rules would describe the behaviour of the Rational Agent in response to the communication events, i.e. in response to messages from other agents. Derivation rules correspond to the rules of the E-commerce ontology that are described in section [**Support for business rules by the Rational Agent**]. Application of "vivid agents" in the domain of business rules' management is described in [9].

2.2.2 BDI Agent Architecture

Another popular agent architecture is the so-called BDI (Belief-Desire-Intention) architecture. As the name indicates, BDI agents are characterised by a "mental state" with three components: beliefs, desires, and intentions. Intuitively, beliefs correspond to information that the agent has about its environment. Desires represent "options" available to the agent – different possible states of affairs that the agent may choose to commit to. Intentions represent states of affairs that the agent has chosen and has committed resources to. An agent's practical reasoning involves repeatedly updating beliefs from information in the environment, deciding what options are available, "filtering" these options to determine new intentions, and acting on the basis of these intentions [20].

An instantiation of the BDI-architecture, a Jam-agent [21] is depicted in Figure 2.1.

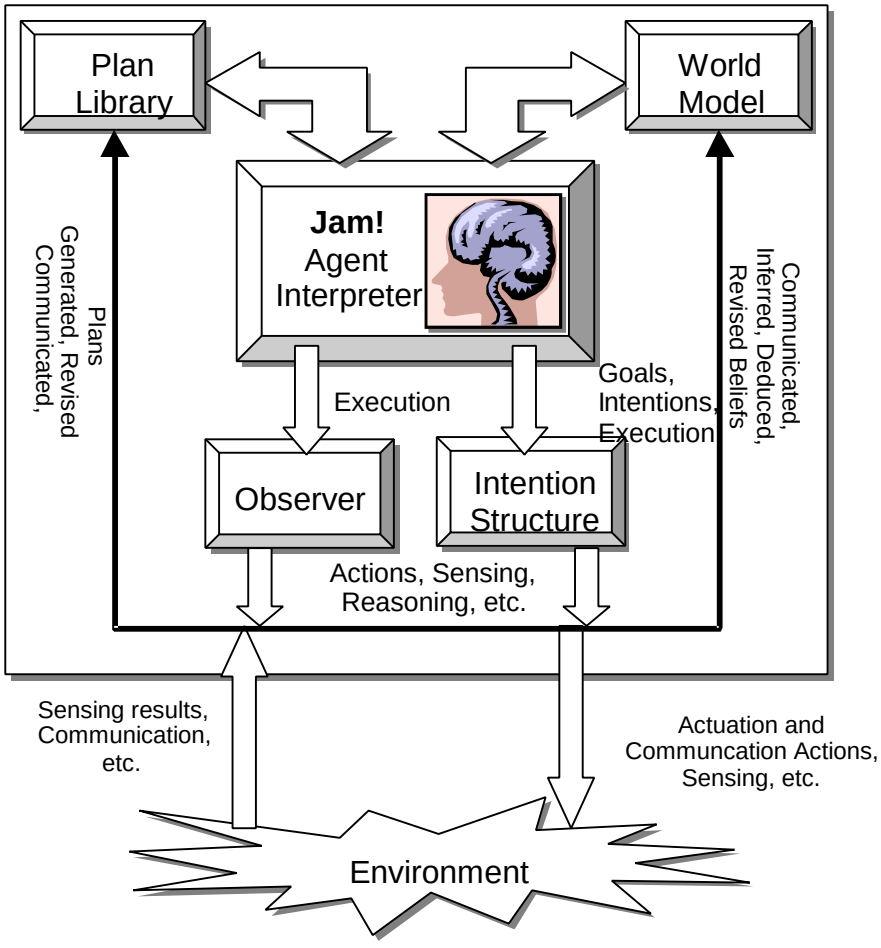

Figure 2.1: The Jam! Intelligent agent Architecture.

As can be seen in Figure 2.1, each Jam agent is composed of five primary components: a *world model*, a *plan library*, an *interpreter*, an *intention structure*, and an *observer*. The world model is a database that represents the beliefs of the agent. The plan library is a collection of plans that the agent can use to achieve its goals. A *plan* of a Jam agent defines a procedural specification for

accomplishing a goal. Its applicability is limited to a particular goal, and may be further constrained to a certain *precondition* and *context* that has to be maintained and to the plan's *utility* expressed as a numeric value. The procedure to follow in order to accomplish the goal is given in the plan's procedural *body*. The interpreter is the agent's "brains" that reason about what the agent should do and when it should do it. The intention structure is an internal model of the goals and activities the agent currently has and keeps track of progress the agent has made toward accomplishing those goals. The observer is a lightweight plan that the agent executes between plan steps in order to perform functionality outside of the scope of its normal goal/plan-based reasoning (e.g., buffer of incoming messages).

According to the Jam execution semantics, changes to the world model or posting of new goals triggers reasoning to search for plans that might be applied to the situation (this list of plans is called the Applicable Plan List, or APL). The Jam interpreter selects one plan from this list of applicable plans and *intends* it (i.e., commits itself to execute the plan). The act of intending the plan places the now-instantiated plan (it has, at this point, a variable binding specific to the current situation) onto the agent's *intention structure*, the agent's multiple-goal runtime stack. The agent may or may not immediately execute the newly intended plan, depending upon the plan's utility relative to that of intentions already on the intention structure.

The agent developer deals explicitly with the world model, the plan library, the observer, and specification of the agent's initial goals. In addition to the primary components described above, there exists a function library and interface for specifying and performing low-level, non-decomposable "primitive" functions.

2.3 Agent Communication Languages

To allow agents to inter-operate, a number of agent communication languages have been designed. An *agent communication language* is based on typed messages. It has to contain message types that represent the most basic communication acts such as:

- supplying new information, e.g. by means of the message type *TELL(fact)*;
- query answering, e.g. by means of the message types *ASK(query)* and *REPLY(answer)*;
- requesting certain actions, e.g. by means of the message types *REQUEST*, *CONFIRM*, and *DISCONFIRM* [19].

In contrast to the application-specific messages in object-oriented programming, message types of an agent communication language provide a more abstract level that can be handled according to a general semantics of communication. Such semantics is based on the philosophical *speech act theory* [13]. In speech act theory, the attitude of the speaker towards the content of a communication act is called the "illocutionary force". Thus, a communication act can be represented as a *typed message* (*performative*) of the form *m[c]* where the message type *m* indicates the "illocutionary force" (such as *TELL*, *ASK*, *REQUEST*, *PROMISE*, etc.) and *c* denotes the message content (such as a proposition or an action) [19].

Agent communication languages, such as KQML [12] and FIPA's ACL [5], provide a set of performatives based on speech acts. Though such performatives can characterise message types, efficient languages to express message content that allows agents to "understand" each other have not been effectively demonstrated. Thus, the *ontology problem* – that of how can agents share meaning – is still open [20]. It is one of the goals of the MKBEEM project to take a step towards solving this problem.

2.4 Commitments and Claims Between Agents

In the field of (electronic) commerce, different commitments and the corresponding claims between parties involved emerge. It is extremely important to represent them explicitly in the virtual knowledge bases of the agents involved. The explicit representation and processing of commitments helps to establish coherent behaviour in an E-commerce system.

According to [9, 19], commitments towards and claims against other agents to perform certain actions or to see that certain conditions hold may arise from certain communication acts of an agent communication language. For instance, sending a sales quotation to a customer commits the vendor to reserve adequate stocks of the quoted item for some specified time. Likewise, acknowledging a sales order implies the creation of a commitment to deliver the ordered items on or before the specified delivery date.

There are two kinds of commitments: commitments to perform an action and commitments to see it that some condition holds. Commitments of other agents towards the agent (e.g. the Rational Agent) under consideration are viewed as claims against them.

Commitment and claim processing includes the creation, cancellation, delegation, and fulfilment of commitments and claims. Commitments have to be fulfilled unless certain exceptional circumstances warrant their cancellation. If a commitment cannot be fulfilled or is otherwise violated, some form of compensation may have to be negotiated. When a commitment is fulfilled, it is discharged. It should be possible to specify these processing steps in a declarative way as business rules (v. section **Support for business rules by the Rational Agent**) in the E-commerce ontologies of the Rational Agent, and make them operational in the form of action rules of the Rational Agent.

2.5 Practical Aspects

In the selection of tools and technologies that will be used in MKBEEM, practical issues must be taken into account as well, given that this is an R&D project meant to deliver working software. Such issues include:

- Availability of off-the-shelf implementations, especially licensing and pricing policies, access to the source code etc.
- Portability, platform and network independence.
- Robustness and quality of the implementation, based on known applications and application domains, previous experiences and results, and wideness of acceptance.
- Scalability, modularity, expandability and efficiency of the implementations.
- Ease of use and deployment, quality of the accompanying documentation and other facilities/tools offered to the programmer (debugging, monitoring etc.).

These trivial factors are certainly not specific to either the project or agent technology. However, they have a special weight in the case of MKBEEM, because a big portion of the technologies under consideration is at an emerging, immature stage. Therefore such practical factors cannot be brushed aside since they are not at all guaranteed to be within acceptable levels. We conclude chapter 2 by stating that our highest priority in the selection of tools is their dependability.

3 Technology Review

In this chapter, we review the most interesting technological approaches to agent applications. The chapter covers standardisation and specification attempts, as well as actual platforms and tools.

There are three main efforts under way for the standardisation of agent systems. These are the FIPA specifications, KQML and MASIF by OMG. In the following sections, we review each of these attempts and some of the existing implementations and tools that support them.

3.1 FIPA Specifications

The Foundation for Intelligent Physical Agents (FIPA) is a non-profit international organisation. Its purpose is the promotion of agent-based technologies. The main effort of FIPA is towards the production of internationally agreed-upon specifications that provide a standard for the development of agent-based applications, services and equipment. FIPA's vision of the future landscape in agent technology depicts large agent societies, in which agents can co-operate. Such multi-agent systems can only be realised if the interoperability of agents in all levels is guaranteed through adequate, widely adopted standards. To achieve this, FIPA is open for membership to everyone and its current members represent more than 50 organisations worldwide.

The first batch of FIPA specifications, titled FIPA97, provide:

- a commonly agreed means by which agents can communicate with each other so they can exchange information, negotiate for services, or delegate tasks
- facilities whereby agents can locate each other (i.e. directory facilities)
- an environment which is secure and trusted where agents can operate and exchange confidential messages
- a unique way of identifying other agents (i.e. globally unique names)
- a means of accessing non-agent and legacy systems, if necessary
- a means of interacting with users

Furthermore, with its second batch of specifications, FIPA98, FIPA addresses the issue of agent migration between platforms.

Not all FIPA specifications are equally pertinent to the MKBEEM project. Those most relevant are the core specifications:

- FIPA97 Spec 1: Agent Management
- FIPA97 Spec 2: Agent Communication Language

In the next sections, we give an overview of these specifications. The complete specifications, as well as other information about FIPA, can be found at its web site, http://www.FIPA.org.

3.1.1 Agent Management System

The Agent Management System is the subject of the first part of the FIPA97 specification. It provides an account of the minimum requirements that an open agent system has to meet, in order to guarantee sufficient management capabilities.

More than that, it outlines a framework in which FIPA-compliant agents can operate seamlessly. The FIPA97 AMS defines the following concepts:

- agent reference model
- agent platform (AP)
- agent domain

Furthermore, it provides a definition of a standard agent management ontology, which can be used to achieve interoperability between disperse FIPA-compliant agent platforms.

3.1.1.1 Fundamental Agents

The agent reference model defines the fundamental entities in an agent system. An agent is defined as *"the fundamental actor on an agent platform which combines one or more service capabilities into a unified and integrated execution model which may include access to external software, human users and communications facilities".* An agent is identified unambiguously by a Globally Unique Identifier (GUID), also known as agent name. It may be registered at a number of addresses at which it can be contacted. Three special types of agents, namely the Directory Facilitator (DF), the Agent Management System (AMS) and the Agent Communication Channel (ACC), support agent management.

- A DF is a "yellow pages" server. Agents may register their services with the DF or query the DF to find out what services are offered by which agents. At least one DF must be resident on each AP (the *default* DF). When queried about a specific service, a DF may contact other DFs to obtain information.
- The AMS is a "white pages" server, by providing a mapping between GUID and contact addresses for the agents. It manages the creation, deletion and general status of the agents in the platform. There should be one AMS for each agent platform.
- The ACC is the message routing agent. It uses the information provided by the AMS to route messages between agents on the same, or on different APs. Concerning the transport mechanism, FIPA97 specifies that ACCs must support at least the Internet Inter-Orb Protocol (IIOP), which is the foundation of CORBA. This does not imply that every other agent has to support IIOP. Agents within a platform are free to use any transport protocol, as long as the platform's ACC supports it. This does not limit their ability to inter-operate with FIPA-compliant agents on different APs, as the ACC provides the necessary interface. Also, direct communication between agents, without the mediation of the ACC, is both possible and allowed. The ACC's role is to provide a standard, omnipresent communication medium, rather than a mandatory one.

3.1.1.2 Agent Platforms and Domains

An Agent Platform (AP) is an infrastructure in which agents are deployed. FIPA-compliant APs comprise an AMS, an ACC and one or more DFs. On the other hand, an Agent Domain (AD) is logical clustering of agents defined by membership in a directory maintained by the domain's DF. Each domain has exactly one DF. Agents may belong to one or more domains. A DF can register with other DFs, thus providing a means for inter-domain information querying.

3.1.1.3 Compliance Requirements

In order to be FIPA AMS compliant, an agent platform must conform to the following requirements:

1. It must contain a DF, an ACC and an AMS. Each of these agents must be able to respond to a set of performatives in ACL, defined in the FIPA specification Part 1.
2. IIOP must be supported as the message transport protocol. Other protocols may be supported optionally.

3.1.2 Agent Communication Language

FIPA's ACL is a universal message-oriented communication language. It defines a standard way to package messages, in order to guarantee that the recipient of the message will be able to understand its purpose. The ACL defines a minimal set of message types, called *performatives,* necessary for agent communication. However, one has to bear in mind that ACL is an all-purpose communication language, hence an application will typically have to use a subset of these performatives.

FIPA claims that this universal approach to agent communication carries a number of advantages, such as [5]:

- dynamic introduction and removal of services
- customised services can be introduced without a requirement to re-compile the code of the clients at run-time
- allow for more de-centralised peer-peer realisation of software;
- a universal message based language approach providing consistent speech-act based interface throughout software (flat hierarchy of interfaces);
- asynchronous message-based interaction between entities.

In essence, an agent's commitment to ACL provides to it the means to flexibly co-operate with other agents and services that do not have to be defined a priori. Intelligent agents based on this philosophy can even exploit services that were not initially foreseen.

3.1.2.1 Compliance Requirements

FIPA defines the minimal requirements that an agent must adhere to, in order to be FIPA ACL compliant [5]:

1. Agents should send *not-understood* if they receive a message that they do not recognise, or if they are unable to process the content of the message. Agents must be prepared to receive and properly handle a *not-understood* message from other agents.
2. An ACL compliant agent may choose to implement any subset (including all, though this is unlikely) of the pre-defined message types and protocols. The implementation of these messages must be correct with respect to the referenced act's semantic definition.
3. An ACL compliant agent that uses the communicative acts whose names are defined in this specification must implement them correctly with respect to their definition.
4. Agents may use communicative acts with other names, not defined in this document, and are responsible for ensuring that the receiving agent will understand the meaning of the act. However, agents should not define new acts with a meaning that matches a pre-defined standard act.
5. An ACL compliant agent must be able to correctly generate a syntactically well-formed message in the transport form that corresponds to the message it wishes to send. Symmetrically, it must be able to translate a character sequence that is well formed in the transport syntax to the corresponding message.

3.1.2.2 ACL Message Description

In a multi-agent system, agents can only achieve their goals through co-operation with other agents, i.e. through *influencing* other agents to perform actions for them. The acts that agents perform to influence the actions of other agents are called *communicative acts.* A communicative act is performed by sending a message denoting that act.

An example of an ACL message is shown in the following figure:

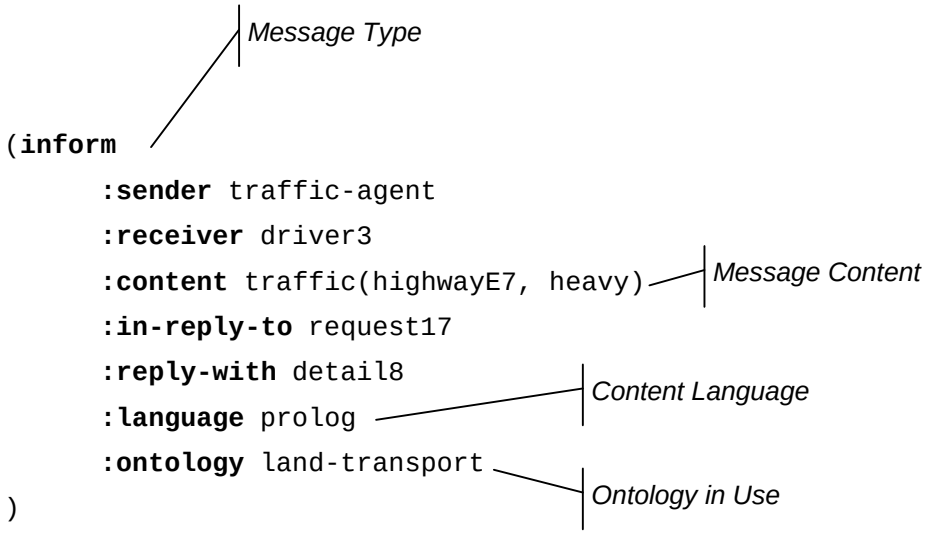

Figure 3.1: An example ACL message

The message structure begins with a word identifying the communicative act, i.e. the message type. The rest of the message is made up of message parameters, beginning with a colon and a keyword identifying the parameter. One of the parameters contains the content of the message, while two others specify the content language and the ontology referenced in the content. These three parameters enable the receiver to understand the message, while the message type tells it what to do

with it. The rest of the parameters help the message transport to deliver the message and the receiver to correctly identify the message as part of a dialogue. User-defined parameters are also supported.

3.1.2.3 FIPA ACL and KQML

Both KQML and ACL are communication languages based on speech act theory, which states that individual communications can be reduced to one of a few primitive speech acts, which shape the basic meaning of that communication. The full meaning is conveyed by the meaning that the speech act itself imparts to the content of the communication. In KQML, the speech act is usually called the *performative*.

KQML was designed originally to fulfil a very pragmatic purpose as part of the ARPA Knowledge Sharing Effort (KSE) consortium. Initially, the semantics of the performatives were described informally by natural language descriptions. Subsequent research has addressed the need for a more precise semantics [14], though it is not clear that the proposed semantics has been universally adopted. As a result, several versions of KQML exist.

In contrast, ACL was designed from its inception [15] to be grounded in a formally defined semantics.

The purpose of KQML is to enable agents to communicate a number of performatives, concerning querying and information passing, as well as managing communications and advertising services. This is an ambitious goal to pursue, while at the same time trying to maintain some economy in the language.

ACL does not attempt to cover all these needs in itself. Indeed, most of the management issues are left to the agent management system. In that aspect it overcomes KQML's dilemma between economy and universality.

3.2 FIPA-compliant Platforms and Tools

FIPA members and non-members are developing or have already developed software that implements one or more FIPA specifications. Some notable attempts have been made by:

- **British Telecom (BT).** BT is developing a FIPA ACL compliant version of **ZEUS**, a Java-based collaborative agent development toolkit. This has gone to trial in late 1998 within BT. Additionally, a number of demonstrator applications have been developed using parts of the FIPA97 specifications. This includes:
  - AEW (Agent-enhanced Workflow) an agent-based process management system that integrates agent technology with a commercial workflow management system.
  - PROTEUS, which supports proactive management of ATM networks using collaborative intelligent agents.
  - PAF (Personal Agent Framework) a unified environment for several personal agent systems in a knowledge management context.
- **Comtec.** Comtec has implemented the ACL parser and generator, interpreters of SL0 and SL2, FIPA-Request and FIPA-Query message types. The implementation is based on Kawa Scheme (a dialect of Lisp written in Java). All the agent management functionality and software integration is implemented.
- **CSELT.** CSELT has made an ACL parser available for non-commercial purposes**. JADE**ä (Java Agent Development kit) is a software development framework aimed at developing multi-agent systems and applications, conforming to FIPA standard for intelligent agents. It includes two main products: a FIPA-compliant agent platform and a package to develop Java agents. JADE has been coded in full Java source code.
- **Fujitsu. AGENTPRO,** Fujitsu's soon-to-be-released agent system software product, is modified to be FIPA97 compliant to conduct interoperability tests. Currently AGENTPRO uses KQML and KIF.
- **GMD FOKUS**. Grasshopper ACL is a FIPA 97 compliant add-on to the Grasshopper mobile agent platform, by IKV++ (http://www.ikv.de/), which is OMG MASIF compliant. The FIPA conformance has been successfully tested with Alcatel and Nortel FIPA platforms in the context of the ACTS FACTS projects. GMD FOKUS is currently working on validating Grasshopper ACL by some applications in the contexts of active VPN service provisioning and management, stock market support and Virtual Enterprise management within some European ACTS/ESPRIT projects. Mobility support based on FIPA 98 specification is under consideration.

- **Kimura Laboratory of The University of Electro-Communications (Denki-Tusin-Daigaku)**, that have produced two FIPA-compliant tools:
  - Falcon, a FIPA97 compliant agent library, where ACL and SL are in use and agents can move across Falcon platforms that contain a DF, AMS and ACC. Components have been coded in pure Java source code.
  - Ascot, a visual programming tool for agent systems and applications, conforming to FIPA97 specification. Components of Falcon have been used in Ascot.
- **Nortel Networks.** Nortel Networks has announced the public availability of a FIPA compliant agent platform, **FIPA-OS**. FIPA-OS is an open source implementation of the mandatory elements contained within the FIPA specification for agent interoperability. In addition to supporting the FIPA interoperability concepts, FIPA-OS also provides a component-based architecture to enable the development of domain-specific agents, which can utilize the services of the FIPA Platform agents. FIPA-OS is an experimental agent framework, originating from research at Nortel Networks Harlow Laboratories in the UK.
- **Siemens.** Siemens have produced an implementation of a FIPA-compliant platform, containing the DF, AMS and ACC and providing parsing/unparsing services for ACL messages.
- **SPAWAR Systems Center**. There are two FIPA-based systems under development by members of the Intelligent Information Exploitation and Retrieval (IIER) project, sponsored by the U.S. Office of Naval Research:
  - TIIERA (Tactical Intelligent Information Exploitation and Retrieval Agents) is currently compliant with FIPA '97. It extends from the FIPA core to include management agents that generate and monitor the execution of complex information-seeking plans. The main objective of the work is to exploit external information sources and applications to obtain filtered and aggregated information tailored to particular users and situations. TIIERA is written in JAVA. The FIPA core is known as IIER_FIPA and is currently being made available to agencies of the U.S. Government and to their contractors.
  - FIPA_SMART (FIPA-based Stationary and Mobile Agent Resource Toolkit) is compliant with FIPA '98. One of its planned uses is as part of the agent management infrastructure for a technology application program called Adaptable Courses of Action. FIPA_SMART is written in JAVA and is available upon request.

We present here a more detailed description of **ZEUS** (British Telecom), **FIPA-OS** (Nortel Networks) and **JADE** (CSELT). Furthermore, **Grasshopper** is covered in the section on OMG MASIF.

3.2.1 JADE

JADE (Java Agent DEvelopment Framework) [7] is a software framework fully implemented in Java language. It simplifies the implementation of multi-agent systems through a middle-ware that claims to comply with the FIPA specifications and through a set of tools that supports the debugging and deployment phase. The agent platform can be distributed across machines (which not even need to share the same OS) and the configuration can be controlled via a remote GUI. The configuration can be even changed at run-time by moving agents from one machine to another one, as and when required.

JADE is completely implemented in Java language and the only system requirement is the version 1.2 of JAVA (the run time environment or the JDK).

Now JADE is distributed in open source software under the terms of the LGPL (Lesser General Public License Version 2). Currently, the latest version of JADE is 1.3.

The goal of JADE is to simplify the development of multi-agent systems while ensuring standard compliance through a comprehensive set of system services and agents in compliance with the FIPA specifications: naming service and yellow-page service, message transport and parsing service, and library of FIPA interaction protocols ready to be used.

The JADE Agent Platform complies with FIPA specifications and includes all those mandatory agents that manage the platform, that is the ACC, the AMS, and the DF. All agent communication is performed through message passing, where FIPA ACL is the language to represent messages.

The agent platform can be split on several hosts. Only one Java application, and therefore only one Java Virtual Machine (JVM), is executed on each host. Each JVM is basically a container of agents

that provides a complete run time environment for agent execution and allows several agents to concurrently execute on the same host.

The communication architecture of JADE offers flexible and efficient messaging, where JADE creates and manages a queue of incoming ACL messages, private to each agent; agents can access their queue via a combination of several modes: blocking, polling, timeout and pattern matching based. The full FIPA communication model has been implemented and its components have been clearly distinguished and fully integrated: interaction protocols, envelope, ACL, content languages, encoding schemes, ontologies and, finally, transport protocols. The transport mechanism, in particular, is like a chameleon because it adapts to each situation, by transparently choosing the best available protocol. Java RMI, event-notification, and IIOP are currently used, but more protocols can be easily added and integration of SMTP, HTTP and WAP has been already scheduled. Most of the interaction protocols defined by FIPA are already available and can be instantiated after defining the application-dependent behaviour of each state of the protocol. SL and agent management ontology have been implemented already, as well as the support for user-defined content languages and ontologies that can be implemented, registered with agents, and automatically used by the framework.

Basically, agents are implemented as one thread per agent, but agents often need to execute parallel tasks. Further to the multi-thread solution, offered directly by the JAVA language, JADE supports also scheduling of co-operative behaviours, where JADE schedules these tasks in a light and effective way. The run-time includes also some ready to use behaviours for the most common tasks in agent programming, such as FIPA interaction protocols, waking under a certain condition, and structuring complex tasks as aggregations of simpler ones.

The agent platform provides a Graphical User Interface (GUI) for the remote management, monitoring and controlling of the status of agents, allowing, for example, to stop and restart agents. The GUI allows also creating and starting the execution of an agent on a remote host, provided that an agent container is already running.

3.2.1.1 JESS Rule Language

Among the others, JADE offers also a so-called JessBehaviour that allows full integration with JESS (The Java Expert System Shell) [8], where JADE provides the shell of the agent and guarantees (where possible) the FIPA compliance, while JESS is the engine of the agent that performs all the necessary reasoning.

JESS is a rule engine and scripting environment written entirely in Sun's Java language by Ernest Friedman-Hill at Sandia National Laboratories in Livermore, CA. JESS was originally inspired by the CLIPS expert system shell, but has grown into a complete, distinct Java-influenced environment of its own. Using JESS, you can build Java applets and applications that have the capacity to "reason" using knowledge you supply in the form of declarative rules. JESS is surprisingly fast, and for some problems is faster than CLIPS itself (using a good JIT compiler, of course.)

The core JESS language is still compatible with CLIPS, in that many JESS scripts are valid CLIPS scripts and vice-versa. Like CLIPS, JESS uses the Rete algorithm to process rules, a very efficient mechanism for solving the difficult many-to-many matching problem (see for example "Rete: A Fast Algorithm for the Many Pattern/ Many Object Pattern Match Problem", Charles L. Forgy, Artificial Intelligence 19(1982), 17-37.) JESS adds many features to CLIPS, including backward chaining and the ability to manipulate and directly reason about Java objects. JESS is also a powerful Java scripting environment, from which you can create Java objects and call Java methods without compiling any Java code.

As such, JESS is very well suited for representing the rules of E-commerce ontologies described in Section 2.Y. Moreover, a combination of the Rational Agent based on the JADE Agent Platform and E-commerce ontologies described by JESS also enables to enforce the rules described by JESS.

VTT Information Technology is also successfully using JESS in one of its other projects.

3.2.1.2 Evaluation of JADE

JADE is a relatively mature FIPA implementation that was recently released as Open Source. The size of its current version is about 500 kB. JADE relies on RMI for internal communication and on SunIDL for external communication. Its documentation consists of the 51-page Programmer’s Guide, JavaDoc API, and of source code and example agents in Java. The communication language of JADE-based agents is FIPA ACL. JADE contains parsers for ACL, and for the message content language SL0. It also provides a sniffer agent that allows graphical tracing of ACL messages between agents in the

form of a message-sequence graph, and ACL test agent for easy testing of ACL interfaces. The advantages and disadvantages of JADE can be summoned as follows:

Advantages:

- FIPA-compliant;
- Java-compliant;
- Free and Open Source;
- Agent framework;
- Nice GUI;
- ACL debugging agents;
- Integration of the JESS Rule Language.

Disadvantages:

- JADE does not include any security or reliability mechanisms;
- Platform IORs must be distributed manually.

3.2.2 FIPA OS

FIPA-OS [22] is an open source implementation of the mandatory elements contained within the FIPA specification for agent interoperability. FIPA-OS is an experimental agent framework, originating from research at Nortel Networks' Harlow Laboratories in the UK.

The primary aim of FIPA-OS is to reduce the current barriers in the adoption of FIPA technology by supplementing the technical specification documents (available at http://www.FIPA.org) with managed open source code. It is envisaged that the quality and functionality of this source code will improve by managing its evolution in the public domain, providing mutual benefit to all adopters and enabling progress in the agent paradigm.

FIPA-OS is based on the FIPA Reference Model depicted in **Figure 3.2**. The FIPA Reference Model shown in **Figure 3.2** illustrates the core components of the FIPA-OS distribution. The agent reference model provides the normative framework within which FIPA Agents exist and operate. Combined with the Agent Life cycle, it establishes the logical and temporal contexts for the creation, operation and retirement of Agents.

The Directory Facilitator (DF), Agent Management System (AMS) and Agent Communication Channel (ACC) are specific types of agents, which support agent management. The DF provides "yellow pages" services to other agents. The AMS and ACC support inter-agent communication. The ACC supports interoperability both within and across different platforms. The Internal Platform Message Transport (IPMT) provides a message routing service for agents on a particular platform, which must be reliable, orderly and adhere to the requirements specified in FIPA 97 V2, Part 2, Agent Communication Language [5].

The ACC, AMS, IPMT and DF form what will be termed the Agent Platform (AP). These are mandatory, normative components of the model. For further information on the FIPA Agent Platform see FIPA 97 V2, Part 1, Agent Management and FIPA 98, Part 13, FIPA 97 Developer's Guide.

In addition to the mandatory components of the FIPA Reference Model, the FIPA-OS distribution includes an Agent Shell, an empty template for an agent. Multiple agents can be produced from this template, which can then communicate with each other using the FIPA-OS facilities.

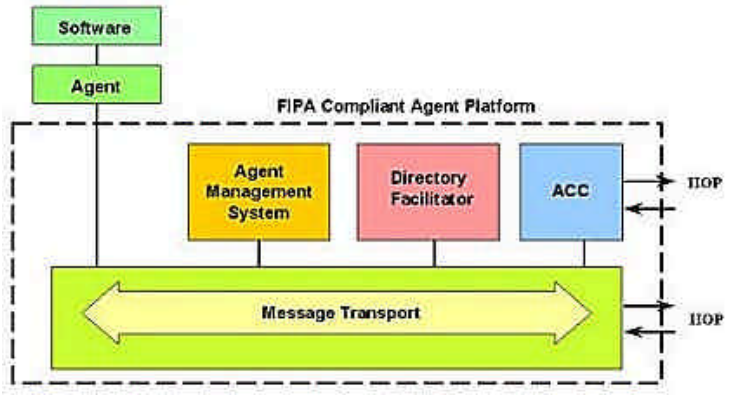

Figure 3.2: The FIPA Reference Model

3.2.2.1 Evaluation of FIPA-OS

FIPA-OS is an Open Source Implementation of FIPA-97. The size of its current version (1.03) with the required 3rd party software is about 1 MB. FIPA-OS relies heavily on 3rd party software. In particular, it makes use of:

- Voyager or Sun IDL for external communication;
- Java RMI for internal communication;
- IBM's XML parser and W3C SiRPAC RDF parser for representing and processing agent profiles.

The documentation of FIPA-OS consists of the 34-page Distribution Notes, JavaDoc API, and of source code and guidelines for add-ons by others. The source of FIPA-OS hasn't just been published, there is some real effort towards getting code contributions. The agent communication language supported by FIPA-OS is FIPA ACL. FIPA-OS contains parsers for ACL, and for the message content language SL0. It also supports platform IORs distributed via a Web server and has a customisable transport layer. FIPA-OS makes use of Sun IDL. The advantages and disadvantages of FIPA-OS can be summarised as follows:

Advantages:

- FIPA-compliant;
- Java-compliant;
- Free and Open Source;
- Agent profiles and content language expressed as XML/RDF;
- Distribution of IORs using a Web server;
- Customisable transport layer.

Disadvantages:

- Complex installation;
- 3rd party software required;
- Poor documentation;
- No reliability or security.

3.2.3 ZEUS

ZEUS is a BT (British Telecommunications) Labs (http://www.labs.bt.com/projects/agents/ZEUS/) advanced development tool-kit for constructing collaborative agent applications. ZEUS is a culmination of a careful synthesis of established agent technologies, with the addition of some new ones, which

provides an integrated environment for the rapid software engineering of collaborative agent applications.

3.2.3.1 The ZEUS Design Philosophy and Main Issues

The aim of the ZEUS project was to facilitate the rapid development of new multi-agent applications by abstracting into a toolkit the common principles and components underlying most existing multi-agent systems. The idea was to create a relatively general purpose and customisable, collaborative agent building toolkit that could be used by software engineers, with only basic competence in agent technology, to create functional multi-agent systems. Thus it defines a multi-agent system design methodology, supports this methodology with an environment for capturing user specification of agents, and automatically generates the executable source code of the user-defined agents. The design philosophy of ZEUS encapsulates the following principles:

- The toolkit should clearly delineate between *domain-level* problem solving and *agent-level* functionality. The latter covers the application-independent multi-agent issues such as communication, co-ordination, task execution and monitoring, exception handling, etc. while the former covers the acquisition, representation and use of domain-specific knowledge in problem solving. The intention being with the agent-level functionality provided, the developers could concentrate on implementing the domain-specific problem solving abilities of their agents.
- Use of the toolkit should be based on the 'visual programming' paradigm. Hence the toolkit would support the agent creation process by providing structured menus and tables that would enable application developers to configure the functionality and modalities required of their agents as simply as possible.
- The toolkit should support an open design to ensure it is easily extensible. Thus, expert users should be able to easily add to the library of agent level components, and configure new agents using a combination of user-defined and system-supplied components.
- The toolkit should utilise ‘standardised’ technology wherever feasible, or be designed with standardisation in mind. Standardisation is essential for the industrial uptake of agent technology. This way, components which could later be replaced with little difficulty by ‘more standardised’ components are used.

In the following paragraphs we present a review of the main techniques proposed by ZEUS for addressing main issues of knowledge level multi-agent systems interoperation such as the *information discovery, communication, ontology, co-ordination* and *legacy software problems.*

- **Information Discovery**: This is typically handled using special-purpose utility agents such as *nameservers* and *facilitators* that function as society-wide white pages (address books) and yellow pages, providing a look-up service for agent’s addresses and abilities respectively. Thus, agents only need to register their address with a nameserver and their abilities with a facilitator to become visible to the society. For scalability and robustness, these utility agents might be arranged in hierarchies similar to that of Internet domain nameservers.
- **Communication**: Concerning the agent communication language (ACL), ZEUS has adopted one of the most 'standardised' agent communication languages currently available, the FIPA ACL, which uses FIPA SL [5] as a content language.
- **Ontology**: Before implementing any agents the application ontology must be defined. The tool used to enter this information is the ZEUS Ontology Editor, or alternatively, an existing ontology can be imported.
- **Co-ordination**: Co-ordinating the behaviour of multi-agent systems is an active area of research with many techniques in use. The main approaches can be broadly classified as organisational structuring, contracting, multi-agent planning, and negotiation. In organisational structuring the prior defined structure of the society (that is, the roles of the different agents and their relationships with one another) is exploited for co-ordination, as typified by client-server systems. ZEUS supports all these forms of agents.
- **Integration with Legacy Software**: As agents are not intended as replacements for legacy software, they must be able to interact with it. Generally speaking there are three possible approaches: *the software could be rewritten*, but this is a costly approach. Alternatively a separate piece of software called a *transducer* could be employed to act as an *interpreter* between the agent communication language and the native protocol of the legacy system. Or thirdly, the *wrapper*

technique could be used to augment the legacy program with code that enables it to communicate using the inter-agent language. ZEUS agents can act as wrappers.

3.2.3.2 The ZEUS Toolkit

The ZEUS toolkit consists of a set of components, written in the Java programming language, that can be categorised into three functional groups (or libraries) an agent component library, an agent building tool and a suite of utility agents comprising nameserver, facilitator and visualiser agents. In the following subsections the ZEUS agent component library, the agent building approach and its associated environment and the suite of utility agents are described.

The Agent Component Library

The Agent Component Library is a collection of classes that form the building blocks of individual agents. Together these classes implement the application-independent *agent-level* functionality required of collaborative agents. The contents of this library address the communication, ontology and co-ordination (or social interaction) issues.

For **communication** the Agent Component Library provides:

- a performative-based agent communication language, in our case FIPA ACL;
- an asynchronous socket-based message passing system;
- an editor for describing domain-specific ontologies — the domain concepts that are defined using the ontology editor are used as part of the content language within the ACL; and
- a frame-based knowledge representation language for representing domain concepts.

For **reasoning and multi-agent co-ordination**, the Agent Component Library provides:

- a general purpose planning and scheduling system suitable for typical task-oriented application domains, and the co-operative problem-solving inherent to these applications, and
- a co-ordination engine that controls the social behaviour of an agent, i.e. when and how it interacts with other agents and the types of contracts it sets up with them.

The **functioning of the planner** and **co-ordination engine** are influenced by the agent's knowledge context, i.e. its available resources and competencies, its organisational relationships with other agents and its available co-operation strategies. Thus, to support these two components, the Agent Component Library also provides:

- a library of predefined re-usable co-ordination protocols, e.g. *contract-net* and various *auction protocols*.
- a number of predefined organisational relationships. The current set of relationships includes *superior*, *subordinate*, *co-worker* and *peer* relations. Agents that are defined as superior to other subordinates agents can delegate tasks to their subordinates. Agents that belong to the same static 'community' can be declared as co-workers, meaning they prefer to interact with one another. The peer relationship is the default, and it does not impose any restrictions on interaction.
- knowledge representation mechanisms and databases for describing and storing the resources and competencies of an agent.

Together, the components of the Agent Component Library enable the construction of an application-independent generic ZEUS agent that can be customised for specific applications by imbuing it with problem-specific resources, competencies, information, organisational relationships and co-ordination protocols. Figure 3.3 shows the architecture of the generic ZEUS agent that is not too dissimilar from other collaborative agent architectures in the literature.

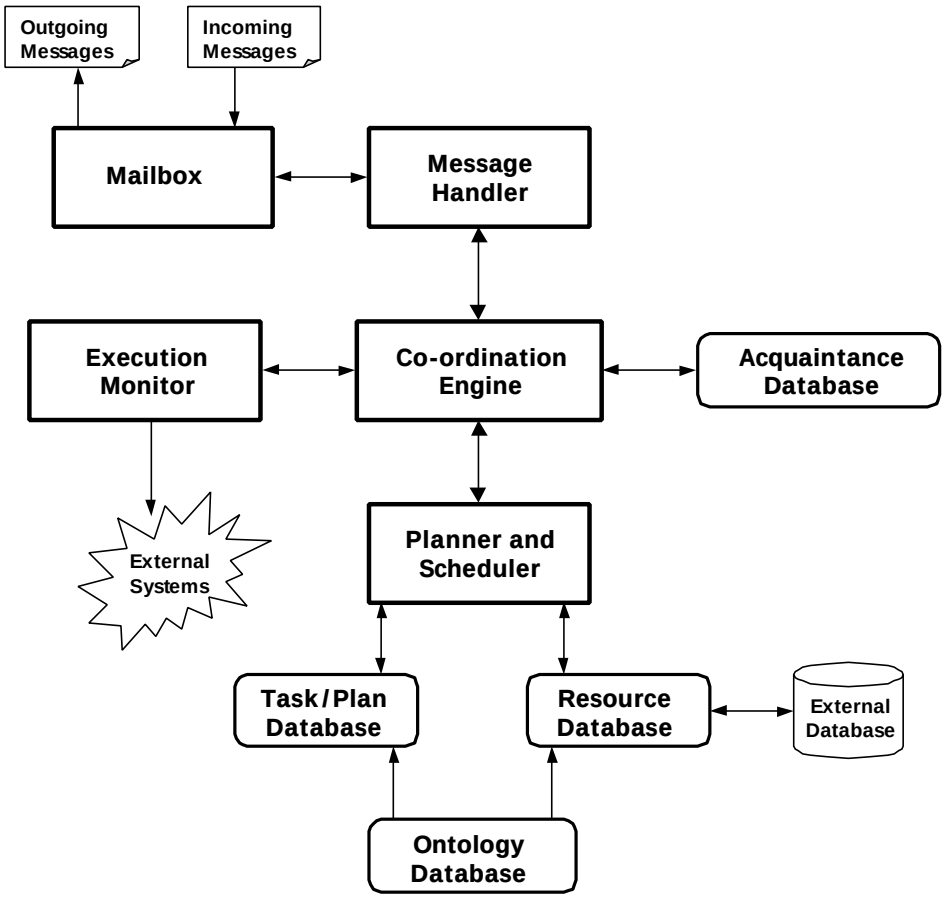

Figure 3.3: Architecture of the generic ZEUS agent

The ZEUS Agent Building Software

Building application-specific agents using the ZEUS toolkit involves configuring the generic ZEUS agent, and equipping it with the necessary for your application functionality. To decrease development time, the ZEUS toolkit provides high-level agent development approach that hides the complexities of the Agent Component Library from the agent developer. The two key aspects are the existence of *an agent creation methodology* which guides the developer through the analysis and design of the intended system, and a *visual* agent development environment figure 3.4

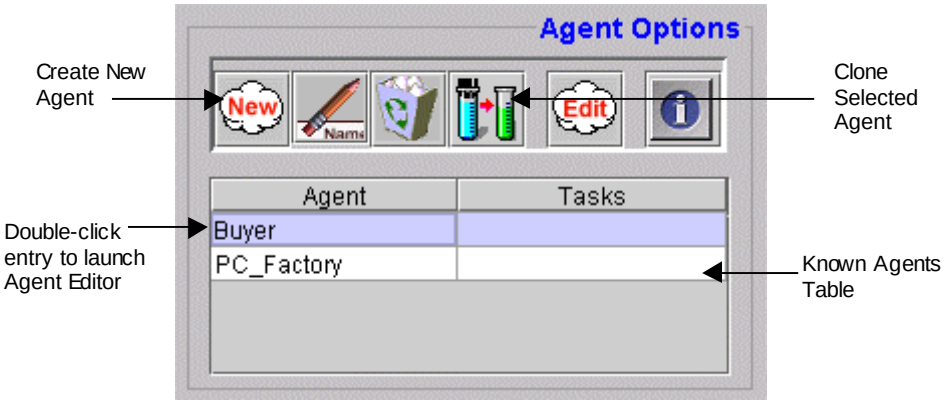

Figure 3.4: Agent Options Panel of the Generator tool.

This following section will briefly present the ZEUS Agent Generator, a suite of integrated editors that support the ZEUS agent design approach. To facilitate ease of use, the editors have been designed to enable users to interactively create agents by visually specifying their attributes. The current suite of editors includes:

An Ontology Editor for defining the ontology items in a domain. Concept categories — referred to as fact templates — can be created for application domains, with the concepts related to one another as appropriate through object-oriented style inheritance and/or composition. Fact objects are defined in terms of their attributes and the valid value ranges for each attribute. Attribute values can be primitive types, lists, other facts or constraint expressions that should ultimately resolve into a primitive type, list or fact.

A Fact/Variable Editor for describing specific instances of facts and variables, using the templates created using the Ontology Editor.

An Agent Definition Editor for describing agents logically. This involves specifying each agent's tasks, its initial resources, and the dimensions of its plan diary.

A Task Description Editor for specifying the attributes of tasks and for graphically composing summary tasks.

An Organisation Editor for defining the organisational relationships between agents, and agents' beliefs about the abilities of other agents.

A Co-ordination Editor for selecting the set of co-ordination protocols with which each agent will be equipped.

The ZEUS Utility Agents

Agent societies have an infrastructure of utility services. The ZEUS suite of utility agents consists of a *nameserver*, a *facilitator* agent and a *visualiser* agent. A ZEUS agent society may contain any number of these utility agents, with at least one nameserver agent. All three utility agents are constructed using the components from the Agent Component Library, and are simplifications of the generic ZEUS agent.

Nameserver agents have only a Mailbox and Message Handler, the components needed for receiving and responding to agents' requests for the addresses of other agents. In addition, nameserver agents maintain a society-wide clock; thus, on initialisation, an agent registers with a nameserver and synchronises its internal clock to that of the nameserver. However, although a society may contain multiple nameserver agents, only the very first one defines *time-zero*.

Facilitator agents have a Mailbox and Message Handler for receiving and responding to queries from agents about the abilities of other agents, and an Acquaintance Database for storing the abilities of the agents. They function by periodically querying all the agents in the society about their abilities, and storing the returned information in their Acquaintance Database. Also, individual agents might advertise their abilities to facilitators. Thus, when an agent wants to find other agents that have a particular competence, they can simply send an appropriate query message to a facilitator agent.

Visualiser agents can be used to view, analyse or debug societies of ZEUS agents. They function by querying other agents about their states and processes, and then collating and interpreting the replies to create an up-to-date model of the agents' collective behaviour. This model can be viewed from different perspectives through visualisation tools supported by the visualiser agents described below:

Society Viewer: shows all the agents in a society, their organisational inter-relationships, and the messages exchanged between the agents during problem solving.

Reports Tool: shows the society-wide decomposition/distribution of active tasks and their execution states.

Agent Viewer: observes and monitors the internal states of the agents.

Control Tool: used to remotely review and/or modify the internal states of individual agents. Thus, an agent's behaviour can be redefined at runtime by using this tool to modify its task, resource, or organisational databases, or even by providing it with new message processing rules and/or co-ordination graphs. In this regard, the control tool is effectively an online version of the Agent Building Software. This tool also facilitates administrative management of agent societies, e.g. agents can be killed or suspended, they can be given news goals, or their old goals can be modified.

Statistics Tool: displays agent and society-wide statistics in various formats.

3.2.3.3 Implementation Details

This section describes the implementation details of the Agent Component library and especially of the main processing components of the generic ZEUS agent:

The Communication Mechanism

Communication between ZEUS agents is via point-to-point TCP/IP sockets, with each message communicated as a sequence of ASCII characters. This is realised through the combined actions of an agent's Mailbox and Message Handler components, permanent threads that run concurrently for as long as the agent is alive. *Why TCP/IP?* TCP/IP was deliberately chosen as the transport protocol for ZEUS messages, in preference to object-oriented middleware solutions like CORBA. As the lowest common denominator, the transport protocol used will ultimately dictate the portability of the agents. Hence the choice was to the most ubiquitous standardised protocol, and for the foreseeable future that is likely to be TCP/IP. It also has the additional advantage of being lightweight, meaning we could implement functionality in the agent layer rather than having to rely on services in the transport layer: the facilitator agent provides a good example. As all aspects of ZEUS agent communication are encapsulated inside the Mailbox, it would be perfectly possible to replace the TCP/IP mechanism with a middleware alternative, should that be felt necessary.

The Language of Communication

Most agent communication languages (ACLs) are based on speech act theory [13], wherein human utterances are viewed as actions in the sense of actions performed in the everyday physical world (e.g. picking up a block). Hence, ACLs specify message types called **performatives**, such as *ask*, *tell*, or *achieve,* which by virtue of being sent from one agent to another, are assumed to effect some illocutionary actions in the receiving agent.

Obviously, inter-agent compatibility will be impossible until all parties adopt the same agent communication language, and fortunately ACL standards do exist. All ZEUS agents communicate using messages that obey the FIPA 1997 ACL specification, which is described in [9].

The Co-ordination Engine

The agent's Co-ordination Engine manages its problem solving behaviours, particularly those involving multi-agent collaboration. In addition to the general requirement for declarative specification of behaviour, the design of the Co-ordination Engine was governed also by the requirement that an agent should be capable of engaging in many tasks simultaneously. This meant that the Engine should support some form of multi-tasking. Simple multi-threading was deemed inappropriate, since the number of independent tasks could potentially run into hundreds. So the choice was to represent problem solving behaviours as recursive transition network graphs, which are interpreted by a recursive finite state machine [5].

The Planner and the Execution Monitor

The role of the Planner/Scheduler is to construct action sequences that achieve desired input goals. The Planner is under the control of the Co-ordination Engine, which initiates planning and also manages the contracting of any sub goals that the agent cannot achieve [5].

Once a plan is constructed and scheduled for execution, each operator in the plan is executed in order, at the operator's scheduled execution time, or alternatively, before the scheduled execution time if there is a free processor available. Operator execution, which is controlled by the Execution Monitor component, involves an invocation of the domain function specified in the operator's specification. The domain function, which is typically some legacy process, is invoked with the operator's preconditions as its input arguments, and it is expected to return the declared effects of the operator. The relevant output of the domain function (i.e. the operator's effects) is then passed as input to the appropriate downstream operators in the plan sequence [5].

Integrating ZEUS Agents with External Programs

ZEUS supports user-defined *Co-ordination Engine graphs* whose nodes make direct calls to external programs. Also the *Resource Database* implements an interface that could be used, for example, with the database connectivity API of Java to link to proprietary databases. For routine problem solving within the declared scope of the ZEUS toolkit, it is expected that for the most part, these primary mechanisms will suffice. However, the Agent Component Library also provides a secondary, more sophisticated interface, although employing it requires significant user programming. This done via a *ZEUSExternal interface class* and an agent internal event model.

The ZEUSExternal interface class allows users to link an external Java class (that implements the interface) to an executing ZEUS agent program. Once linked to the agent program, the external code can utilise the agent's public methods to query or modify the agent's internal state. Thus, for example, the resource and/or plan databases can be queried or modified.

Moreover the internal event model provides a mechanism whereby all significant events occurring in the agent can be monitored. The internal event model is similar to the Java event model and all subsequent events of that type are forwarded to the object. So, using the event model, an external program that is linked to a ZEUS agent can monitor particular events in the agent and react to them.

3.2.3.4 Conclusions

The ZEUS toolkit is an award winning integrated environment for the rapid development of collaborative agent applications. It makes good use of graphical programming, debugging, and visualisation tools, and sets a high standard for integrated GUI-based agent development environments.

As ZEUS is one of the most promising tools under review, we implemented an experimental prototype to obtain hands-on experience with the system. Our experience with the prototype generally justified the impression of a dependable and easy-to use system, but it also surfaced some problems:

- The ZEUS built-in ontology tool does not allow a fact to inherit from two other facts, i.e. multi-inheritance is not allowed. Also, there is no way to import externally created ontologies. This is a severe drawback for our project, since it confines us to the use of the knowledge representation language adopted by ZEUS.
- The task specification is done at build time. This implies that all tasks should be predefined prior to the system's execution. This is not exactly what we would like to have since, in our case, a number of preconditions could dynamically change. This restriction may be eliminated by the use of summary or rulebase tasks.

ZEUS is Open Source software; it is entirely implemented in Java (JDK2) and will run on all major hardware platforms. It is FIPA-ACL compliant and has been undergoing evaluation by researchers and engineers from 30 organisations around the world. ZEUS has been employed successfully as the underlying technology in a number of BT's agent-based R&D projects, e.g., "Agent-based Workflow Management", "Multi-Agent Trading Environment for E-Commerce", and "Agent-based broadband network management". The last version of ZEUS is numbered 1.02.

3.3 KQML

KQML, the Knowledge Query and Manipulation Language, is a language and protocol for exchanging knowledge. It is part of a larger framework, the ARPA Knowledge Sharing Effort, which is aimed at developing techniques and methodology for building large-scale knowledge bases, which are sharable and reusable. KQML is both a message format and a message-handling protocol to support run-time knowledge sharing among agents. KQML can be used as a language for an application program to interact with an intelligent system or for two or more intelligent systems to share knowledge in support of co-operative problem solving.

KQML focuses on an extensible set of performatives, which defines the permissible operations that agents may attempt on each other's knowledge and goal stores. The performatives comprise a substrate on which higher-level models of inter-agent interaction can be developed, such as contract and negotiation nets. In addition, KQML provides a basic architecture for knowledge sharing through a special class of agents called communication facilitators, which co-ordinate the interactions of other agents. The ideas, which underlie the evolving design of KQM, are currently being explored through experimental prototype systems, which are being used to support several testbeds, in such areas as concurrent engineering, intelligent design and intelligent planning and scheduling.

3.3.1 KQML Structure

KQML is conceptually a layered language. KQML expressions consist of a content expression encapsulated in a message wrapper, which is in turn encapsulated in a communication wrapper, as shown in next Figure.

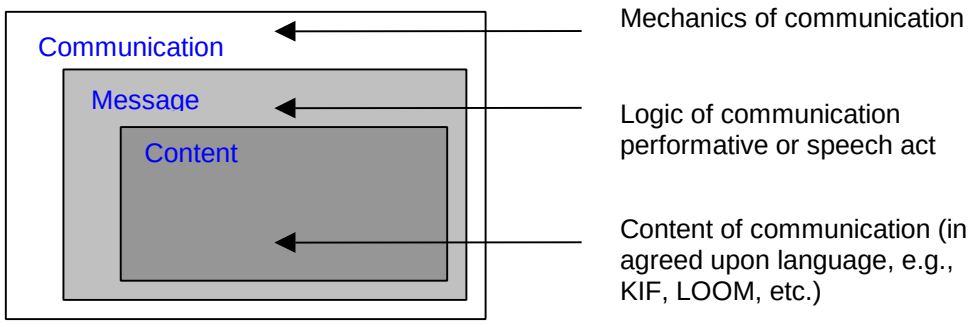

Figure 3.5: KQML Layers.

The **KQML content layer** is the actual content of the message in the program's own representation language. KQML makes no commitments about the format of the content. KQML can carry any representation language, including languages expressed as ASCII strings and those expressed using a binary notation. However, KIF is a good default. The content language that is used in any particular message is specified in the message layer.

The **KQML communication layer** adds features to the message, which describe the lower level communication parameters, such as the identity of the sender and the recipient, and a unique identifier associated with the communication. These are used by the network layer to provide reliable transfer of bytes between processes on a network. The communication is packet oriented and not connection oriented. Hence, agents actually exchange packages.

The KQML communication layer is currently built on TCP/IP, CORBA and e-mail, but could also be built on RPC, UDP or other packet communication media.

The **KQML message layer** forms the core of the language. It is independent of the content language and the communication layer. It determinates the kinds of interactions one can have with a KQML-speaking agent. The primary action of the message layer is to identify the protocol to be used to deliver the message and to supply a speech act or performative which the sender attaches to the content.

The performative signifies that the content is an assertion, a query, a command, or any of a set of known performatives. Because the content is opaque to KQML, the message layer adds a set of features, which describe the content, e.g. the language it is expressed in, the ontology it assumes and the kind of speech act it represents. These features make it possible for the protocol implementation to analyse, route and properly deliver messages.

The KQML message layer is based on an extensible set of well-defined performatives. This is not a required or minimal set; a KQML agent may choose to handle only a few (perhaps one or two) performatives. However, an implementation that chooses to implement one of the reserved performatives must implement it in the standard way. A community of agents may choose to use additional performatives if they agree on their interpretation and the protocol associated with each.

The KQML **communication protocols** include:

- Simple synchronous transactions, e.g. query/reply, assert/acknowledge
- Multi-purpose performatives, e.g. generators
- Asynchronous replies, e.g. subscriptions, monitoring
- Communication services, e.g. forwarding, broadcasting

Conceptually, a KQML message (package) consists of a performative (performatives), its associated arguments which include the real content of the message, and a set of optional arguments which describe the content and perhaps the sender and receiver. For example, a message representing a query about the price of a share of IBM stock might be encoded as:

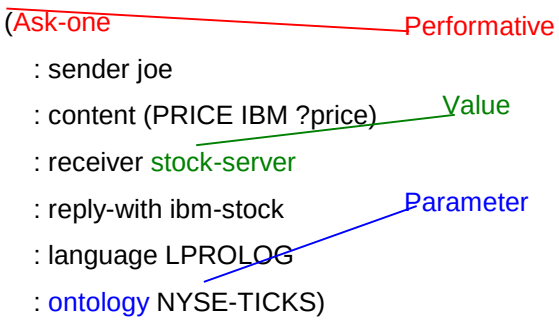

Figure 3.6: - A KQML message example

3.3.2 KQML Facilitation Services

One of the design criteria for KQML was to produce a language that could support a wide variety of interesting agent architectures. KQML's approach to this is to introduce a small number of KQML performatives, which are used by agents to describe the meta-data specifying the information requirements and capabilities and then to introduce a special class of agents called *communication facilitators.* Facilitators are agents that handle knowledge about the information services and requirements of other agents. They can offer services such as forwarding, brokerage, recruiting and content-based routing. Although facilitators are foreseen in KQML, no formal specification for their implementation exists.

3.4 KQML Implementations

The KQML language and implementations of the protocol have been used in several prototype and demonstration systems. The applications have ranged for concurrent design and engineering of hardware and software systems, military transportation logistics planning and scheduling, flexible architecture for large-scale heterogeneous information systems, agent based software integration and cooperative information access planning and retrieval.

There are quite a few implementations of KQML currently available. We list some of them here:

- **JATLite** (Stanford**).** JATLite provides templates written in Java classes for constructing agents. The classes are provided in layers, to be used as desired by the developer. A top-level KQML layer is provided. JATLite, unlike its ancestor JAT, makes no commitment to any special internal agent architecture. JATLite is not a mobile agent platform, but does provide a good infrastructure through which agents can pass messages, whether they are mobile or not.
- **Jackal** (UMBC). Jackal is a Java package, which provides a comprehensive communications infrastructure for Java-based agents. Jackal facilitates the use of the KQML agent communication language, and employs a new, sophisticated protocol for agent naming, addressing and identification (KNS).
- **TKQML/TACKAL** (UMBC). TKQML is a KQML application/addition to TCL/TK, which allows TCL based systems to communicate easily with a powerful agent communication language.
- **MAGENTA** (Stanford). Magenta (C++ version) is an ACL API, which facilitates communication between agents located in a heterogeneous computing environment. Magenta supports communications in agent communication languages, like KQML. Although users of the API can use peer-to-peer communication, the API is especially useful for Anonymous Agent Interaction.
- **JKQML** (IBM). JKQML is a framework and API for constructing Java-based, KQML-speaking software agents that communicate over the Internet. JKQML allows the exchange of information and services between software systems, creating loosely coupled distributed systems. JKQML is based on the 1996 proposal for a new KQML specification.
- **JAFMAS:** JAFMAS provides a framework to guide the coherent development of multi-agent systems along with a set of classes for agent deployment in Java.
- **KAPI** (Lockheed/EIT/Stanford). KAPI supports the passing of KQML messages over the Internet, transparently operating over TCP/IP using KQML string syntax, over MIME multimedia mail (gets around most firewalls), and HTTP (to/from World-Wide-Web browsers and servers).

We review here JATLite and JKQML, which seem to be the more interesting and relevant to MKBEEM.

3.4.1 JKQML

JKQML is a framework and API for constructing Java-based, KQML-speaking software agents that communicate over the Internet [29]. JKQML allows the exchange of information and services between software systems, creating loosely coupled distributed systems. JKQML provides flexibility for the extension of the framework, and it supports the following three protocols:

- KTP (KQML transfer protocol): a socket-based transport protocol for a KQML message represented in ASCII.
- ATP (agent transfer protocol): a protocol for KQML messages transferred by a mobile agent that is implemented using Aglets[2].
- OTP (object transfer protocol): a transfer protocol for Java objects that are contained in a KQML message.

JKQML is based on the 1996 proposal for a new KQML specification.

3.4.1.1 Functionality

The JKQML system provides the following functions:

- Asynchronous message transfer
- Plug-in mechanism for protocol handlers
- Plug-in mechanism for naming services
- Plug-in mechanism for performative handlers
- Plug-in mechanism for interpreters
- Plug-in mechanism for conversation policies
- Multi-threaded conversations
- Persistency for conversations

3.4.1.2 Components

JKQML consists of three major components, the protocol manager, the KQML manager, and the conversation pool. Each component has subcomponents.

The **KQML Manager** provides all the services required to process KQML messages for an agent application. It provides the following services:

- Transferring KQML messages
- Enhancement or customisation of the protocol manager
- Enhancement or customisation of conversation management
- Content interpreters and performative handlers management
- Agents capability management
- Persistency control

The agent application can access other major components through these KQML manager services.

The **Protocol Manager** handles the actual transport mechanism of KQML messages. Currently it supports three transport protocols, one is the KQML Transfer Protocol (KTP), developed for ordinary TCP/IP environment (java.net.Socket), the Agent Transfer Protocol (ATP) for the Aglets environment, and the Object Transfer Protocol (OTP) for transferring Java Objects. Currently OTP is experimental and does not transfer Java byte codes but rather Java objects. Application developers can add their own transport protocol handler or modify the behaviour of protocol handlers, which have already been plugged in.

[2] The Aglets Software Development Kit is an environment for programming mobile Internet agents in Java. Aglets are Java objects that can move from one host on the Internet to another. That is, an aglet that executes on one host can suddenly halt execution, dispatch itself to a remote host, and resume execution there. When the aglet moves, it takes along its program code as well as its data. It is made by IBM and distributed free of charge.

Various transport protocols can be used to transfer a KQML message by installing the appropriate protocol handler plug-in. Ad hoc protocol handler plug-ins for application specific needs can be created and installed into JKQML.

The *naming service* can locate agents by symbolic agent name. Currently a simple agent naming service is supported by JKQML. It provides a mapping between a symbolic agent name and a physical network address (URL). Ad hoc or standard naming (directory) services can be used by creating a specific naming service plug-in.

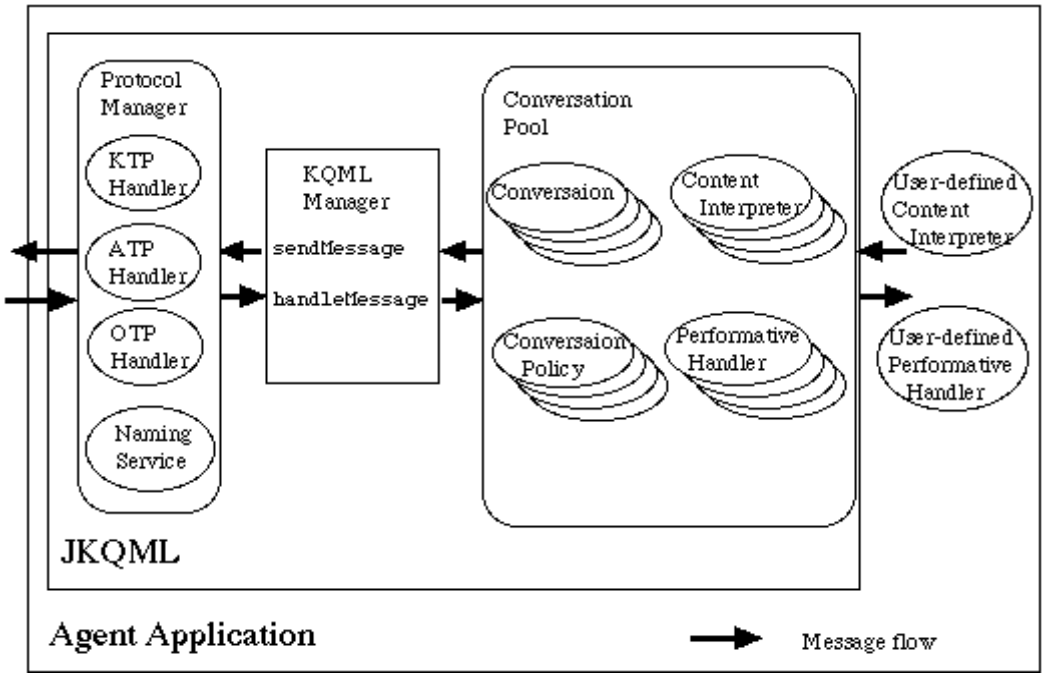

Figure 3.7: The JKQML Architecture

The **Conversation Pool** manages ongoing conversations. It controls message sequences by using conversation policies. The Conversation Pool dispatches received KQML messages to the appropriate performative handler or content interpreter to handle it.

A *conversation* is a sequence of KQML messages that belong to the same thread of communication or interaction between two or possibly more agents. It provides an interface for access to a sequence of KQML messages and results from a performative handler or a content interpreter.

The *conversation policy* is the set of policies or rules that describe the domain of permissible conversations among KQML-speaking agents. Ad hoc conversation policies can be defined and installed into JKQML to control application specific KQML message sequences.

The *content interpreter* handles an incoming KQML message's content. Its function is to translate the content of the incoming message into a suitable internal representation. An incoming KQML message is dispatched to the appropriate content interpreter determined by the language and ontology parameters of the message. User-defined content interpreters can be installed into JKQML. A content interpreter can be defined to process as many performatives as it needs.

The *performative handler* handles an incoming KQML message's content. It uses the information generated by the content interpreter to serve the incoming message. An incoming KQML message is dispatched to appropriate performative handler specified by using language and ontology parameter of the messages. User-defined performative handler can be installed into JKQML.

JKQML supports 22 performatives (message types) of proposed draft new KQML specification.

Two preliminary ontologies are supported. One is the White Pages Ontology for mapping between a symbolic agent name and a physical network address. The other is the Yellow Pages Ontology, which provides a search facility for the addresses of various services.

3.4.2 JATLite

JATLite (Java Agent Template, Lite) is a package of programs written in the Java language that allow users to quickly create new software "agents" that communicate robustly over the Internet [30]. JATLite

provides a basic infrastructure in which agents register with an Agent Message Router facilitator using a name and password, connect/disconnect from the Internet, send and receive messages, transfer files, and invoke other programs or actions on the various computers where they are running.

JATLite facilitates especially construction of agents that send and receive messages using the emerging standard communications language, KQML. The communications are built on open Internet standards, TCP/IP, SMTP**,** and FTP. However, developers may easily build agent systems using other agent languages using JATLite.

JATLite provides a *template* for building agents that utilize a common high-level language and protocol. This template provides the user with numerous predefined Java classes that facilitate agent construction. Furthermore, the classes are provided in layers, so that the developer can easily decide what classes are needed for a given system. For instance, if the developer decides not to use KQML, the classes in the KQML layer are simply omitted. However, if that layer is included, parsing and other KQML-specific functions are automatically included.

JATLite does not endow agents with specific capabilities beyond those needed for communication and interaction. In particular, JATLite does not, by itself, construct “intelligent agents” that seek information or automate human tasks, as discussed in the Artificial Intelligence community. The developer is left free to use whatever theories and techniques are best suited for the targeted application or research**.**

However, it does provide a robust substrate for building such intelligent agents. The JATLite packaged infrastructure allows agents to be portable, to move from one machine to another, and to connect and disconnect from the Internet with automatic queuing and buffering of incoming messages. These features found to be necessary for robust agent behaviour in projects where software agents occasionally fail or migrate, is provided by the Agent Message Router (AMR) infrastructure.

This unique facility also overcomes Netscape security restrictions on applets, allowing them to be full-fledged but highly migratory lightweight agents. Applet agents can be run from any browser: there is no need for any specialized “docking” software to be installed.

3.4.2.1 JATLite Intended Uses

A primary application of JATLite is to “wrap” existing programs by providing them with a front-end that allows them automatically to communicate with other programs, sending and receiving messages, files, etc. JATLite allows Agents to make a single connection to an Agent Message Router (AMR) that buffers messages and handles all IP address information. JATLite provides many useful features. Any Agent sending KQML can use the JATLite if the agent is consistent with JATLite assumptions.

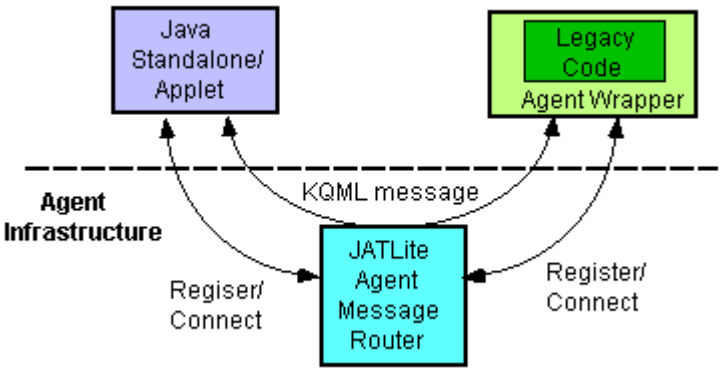

Figure 3.8: JATLite approach to wrapping legacy software

A key idea is that even if no new autonomous agents are ever built, agent technology provides a ”glue” for composing legacy software. JATLite provides standard software for agent communications. KQML and other agent protocols provide standards for message exchange. Various process analysis systems can then provide guidance for the composition of specific messages among the system agents.

It is particularly easy to integrate JATLite with other Java software, but JATLite agents have also been integrated with C++ and Lisp code, albeit without the platform-independent advantages of pure Java.

The basic capabilities of JATLite are:

- Modular construction consisting of layers, each of which can be exchanged with other technologies without affecting the operation of the rest of the package.

- Low-level communications based on TCP/IP, as supported by commonly used operating systems (e.g., Unix, Windows, MAC OS...). Other protocols (e.g., email) can be added easily.
- Agent messages based on the KQML language and protocol, with built-in parsing for the outer layer of messages. The inner “contents” of messages can be in any language (e.g., SQL, Express, KIF).
- Multi-threaded operation, with multiple server sockets and message receiver sockets. Socket connections are persistent and have timeout provisions.
- Provides Message Routers for agent registration, connection, name, and password services.
- Provides storage and queuing of messages for mobile and sporadic agents.
- Supports stand-alone agents in Java and C++, and applet agents through popular WWW browsers (e.g., Netscape, Internet Explorer).
- Built-in FTP capability.

3.4.2.2 JATLite Architecture

The JATLite **architecture** is organised as a hierarchy of increasingly specialised layers, so that developers can select the appropriate layer from which to start building their systems. Thus, a developer who wants to utilise TCP/IP communications but does not want to use KQML can use only the Abstract and Base layers as described below.

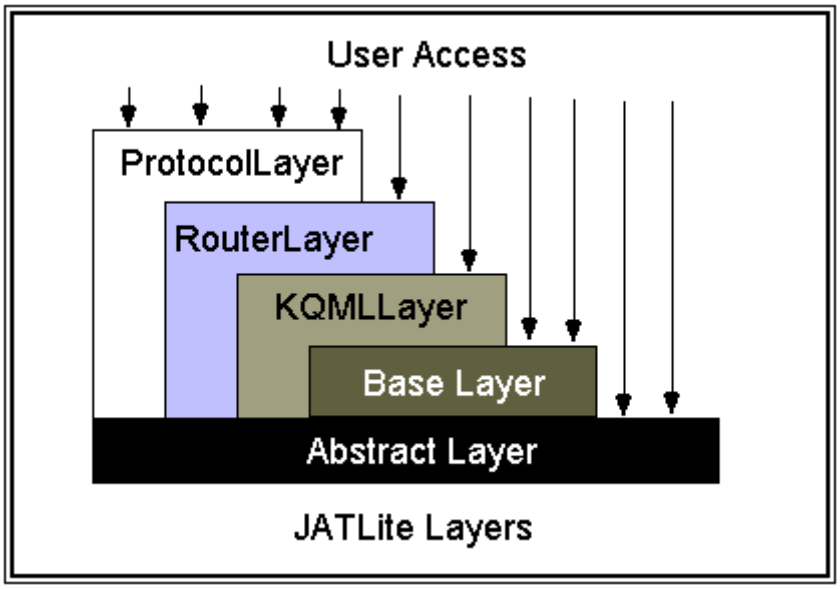

Figure 3.9: JATLite is built as a series of increasingly specialised layers

The ***Abstract Layer*** provides the collection of abstract classes necessary for JATLite implementation. Although JATlite assumes all connections to be made using TCP/IP, one can implement different protocols such as UDP by extending the Abstract Layer.

The ***Base Layer*** provides basic communication based on TCP/IP and the abstract layer. There is no restriction on the message language or protocol. The Base Layer can be extended, for example, to allow inputs from sockets and output to files. The Base Layer can also be extended to provide agents with multiple message ports, etc.

The ***KQML Layer*** provides for storage and parsing of KQML messages. Extensions to the KQML standard, proposed by the Stanford Center for Design Research are implemented to provide a standard protocol for registering, connecting, disconnecting, etc.

The ***Router Layer*** provides name registration and message routing and queuing for agents. All agents send and receive messages via the Router, which forwards them to their named destinations. When an agent intentionally disconnects, or accidentally crashes, the Router stores incoming messages until the agent is reconnected. The Router is particularly important for applet agents, which can only initiate socket connections with the host that spawned them, due to WWW and Java security restrictions.

On top of the Router Layer, the ***Protocol Layer*** will support diverse standard Internet services such as SMTP, FTP, POP3, HTTP etc, both for stand-alone applications and applets. Current beta version supports SMTP and FTP but other protocols can be easily extended from Protocol Layer. If agents are expecting to transfer non-sentential, lengthy data or your agents need to send KQML message through email, Protocol Layer will be a good starting point.

3.5 OMG MASIF

Since the start of intensive research activities in the early 1990, mobile agent technology has gained momentum in various application areas, such as electronic commerce, workflow management, and telecommunications. The reasons for this are several benefits of this rather new technology, such as asynchronous task execution, reduction of network traffic, robustness, distributed task processing, and flexible on-demand service provision. However, legacy technologies, such as the traditional client/server paradigm, are still considered as suitable solutions for many distributed application scenarios. Currently, effort is spent to combine these two approaches in order to realise a unified distributed middleware, supporting both client/server-based remote procedure calls and mobile agents. The recent OMG work on a Mobile Agent System Interoperability Facility (MASIF) [33] specification can be regarded as a milestone on the road toward such a unified middleware, which enables technology- and location-transparent interactions between static and mobile objects/agents.

Today, the CORBA standards gain high acceptance in the world of distributed computing. Various service specifications have been developed, providing standardised, implementation-independent interfaces and a common protocol, and thus enabling a high degree of interoperability between applications of different manufacturers. In contrast to this, mobile agent technology is driven by a variety of different approaches regarding implementation languages, protocols, platform architectures and functionality. In order to achieve a sufficient integration with CORBA, a standard is required for mobile agent technology. This standard has to handle interoperability between different agent platforms and the usability of (already existing) CORBA services by agent-based components.

3.5.1 MASIF Architecture

OMG issued a Request for Proposal (Common Facilities RFP3) [34] for a mobile agent standard in November 1995. The corresponding Mobile Agent System Interoperability Facility (MASIF) submission, developed by Crystaliz, General Magic, GMD FOKUS, IBM, and The Open Group, has been adopted by the OMG in February 1998. The first MASIF-compliant agent platform is Grasshopper from IKV++ [35], and is commercially available.

The idea behind the MASIF standard was to achieve a certain degree of interoperability between mobile agent platforms of different manufacturers without enforcing radical platform modifications. MASIF is not intended to build the basis for any new mobile agent platform. Instead, the provided specifications shall be used as an "add-on" to already existing systems. The following list comprises the mandatory requirements that were identified within the MASIF RFP:

- Marshalling and un-marshalling of agent programs.
- Encoding of agent containers for transport.
- Transport of agents from one agent facility (i.e. execution engine) to another.
- Runtime registration and invocation of agent facilities.
- Runtime query of a named agent facility by agents.
- Runtime security of agents.

Additionally, several optional requirements were defined by the RFP, covering among others the identification and localisation of agents, the starting, stopping, and suspending of an agent's execution, and the runtime monitoring of agent facilities and agents via their names or characteristics.

As shown in Figure 3.10, MASIF has adopted the concepts of agent systems (i.e. agencies), places and regions that are also used by several existing agent platforms.

A *place* groups the functionality within an agency, encapsulating certain capabilities and restrictions for hosted agents. Each agency comprises at least one place in which the hosted agents are running.

A *region* facilitates the platform management by grouping sets of agencies that belong to a single authority.

Two interfaces represent the core of the MASIF standard: The *MAFAgentSystem* interface is associated with every MASIF-compliant agency and provides operations for the management and transfer of agents. The *MAFFinder* interface is associated with a region, i.e. a set of agencies. It is part of a region registration component that supports the localisation of agents, agencies, and places in the scope of a region. As part of a MASIF-compliant agency, a *MAFAgentSystem* object interacts internally with agency-specific services and provides the associated CORBA interface to external users (Figure 3.10). In this way it is possible to communicate with an agency either in a MASIF-compliant way (using the *MAFAgentSystem* interface) or in a platform-specific way (using platform-specific interfaces that may provide additional functionality, not handled by MASIF).

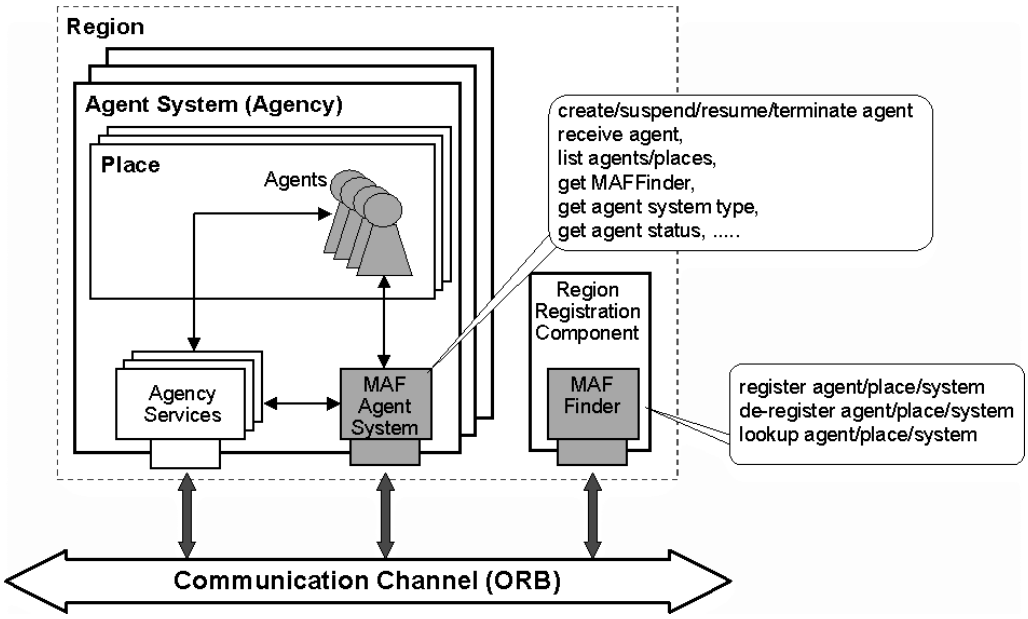

Figure 3.10 MASIF Compliant Platform Architecture

Among others, the following functionality is covered by the MASIF-compliant interfaces due to the requirements defined in the MASIF RFP:

Agent Management

This topic comprises the creation, termination, suspension, and resumption of agents. The *MAFAgentSystem* interface provides several methods for this purpose, i.e. the methods *create_agent*, *terminate_agent*, *suspend_agent*, and *resume_agent*.

Agent Tracking

Agencies, places, and agents are registered in a region registration component via the *MAFFinder* interface. While agencies and places are registered only once during their entire lifetime, mobile agents are de-registered before each migration and registered again (at their new location) after migration. In this way, the region registration component knows the current location of each agent at any time. This location information can be retrieved via lookup methods provided by the *MAFFinder* interface.

Agent Transport

The *MAFAgentSystem* interface offers two methods to support agent migration, i.e. the method *receive_agent* for transferring an agent's state and other required data, and the method *fetch_class* for additionally retrieving an agent's code, if needed.

Agent and Agency Naming

Standardised syntax and semantics of agent and agency names enable agents and agencies to identify each other and allow clients to identify agents and agencies.

Agency Type and Location Syntax

Agency types provide information about important aspects of specific agencies, such as the used implementation language. Before an agent can migrate, it has to determine if the desired destination agency supports the agent's execution. The location is standardised in order to enable agencies to locate each other.

Additionally Covered Standardisation Topics

Apart from the agent-specific CORBA interfaces *MAFAgentSystem* and *MAFFinder,* existing CORBA Common Services, i.e. the Naming, Life Cycle, Externalisation, and Security Service, can be used by agent-based components to enhance the provided functionality. Especially the Security Service is handled in detail, illustrating the problems associated with mobile objects.

3.5.2 General Considerations

Regarding mobile agent technology, the following problem occurs: A mobile agent is by definition able to migrate from one physical network location to another. For its execution, a mobile agent needs a certain runtime environment, i.e. an agency. In order to enable an agent to run in a specific agency, the agent must fulfil various preconditions. For instance, the agent must be implemented with the same programming language as the agency or at least with a language that can be interpreted by the agency. Besides, the agent must know how to access the internal services and resources offered by the agency. Such object-internal aspects are usually not handled by CORBA. This means, the interoperability between different MA platforms is automatically limited if only implementation-independent aspects are covered by the standard, such as done in the context of MASIF.

Since most of today's mobile agent platforms are implemented in Java, it seems to be suitable for new OMG MA standards to define implementation-specific (i.e. Java-specific) conventions that allow a mobile agent to migrate to any standard-compliant agency, independent of its type and manufacturer. This would be the highest degree of interoperability that can ever be achieved. In contrast to this, the *current* MASIF standard does not guarantee that an agent developed for the MASIF-compliant platform A is in any case able to run in an agency of the MASIF-compliant platform B.

3.5.3 Grasshopper

Grasshopper is an industrial platform product, providing functionality for the development and execution of mobile and intelligent agent applications. The Grasshopper platform is written in Java. Presently it is the only agent platform available, which is compliant to both the OMG MASIF (using a MASIF compatibility add-on) and the FIPA ACL specifications (similarly, using the Grasshopper ACL add-on).

3.5.3.1 Architecture Overview

The root concept in Grasshopper architecture is the *agency* [31]. An agency is a platform on which Grasshopper agents reside. An agency is realised as a Java process, running on its own Java Virtual Machine (JVM). It provides the required functionality for supporting the execution and management of the hosted agents, including a communication service, security service, persistence service, and access to a management API and an agency domain service. Besides, an agency provides graphical and/or textual user interfaces for administration purposes.

Agents always reside in a particular *place* in an agency. A place is a logical entity inside a Grasshopper agency. Each agency has at least one place, named 'InformationDesk'. Additional places can be added during the agency's start-up or later during its runtime. A user can define a security policy for each place separately.

Agents in Grasshopper are implemented as Java classes. At runtime, each agent has its own thread running inside the parent agency's process. Agents have two fundamental binary properties, distinguishing four basic agent classes. These properties are *mobility* and *persistence.* The four agent classes produced by combination of these properties are depicted in the following figure.

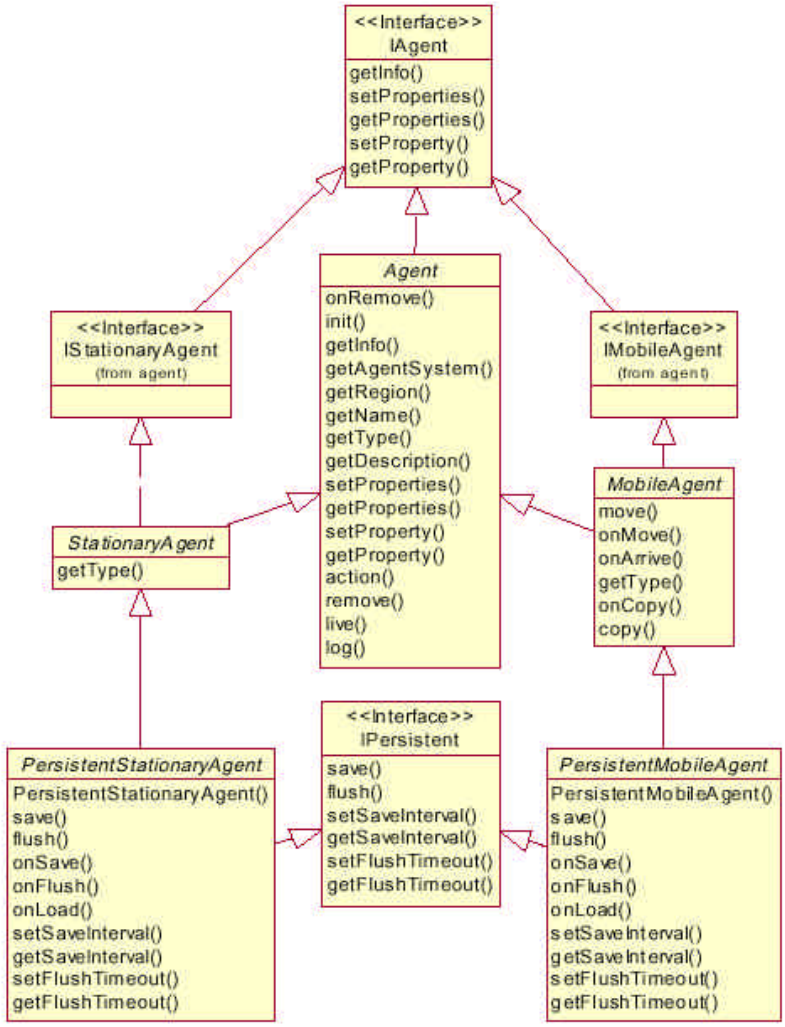

Figure 3.11: The Grasshopper Agent Classes

Mobile agents are able to migrate from one place to another, either within or across agencies, while stationary agents reside in their creation place for their entire lifecycle. The second decision when implementing a Grasshopper agent is whether the agent shall be recoverable after a system crash. Grasshopper provides a persistence service in order to enable agencies to persistently store an agent's data state within the local file system. In case of a system crash or simply the termination of an agency, all persistently stored agents can be recovered when the agency is restarted.

3.5.3.2 Agent Management Functionality

Agent management is the responsibility of agencies. An agency consists of two discrete parts: the core agency and one or more places. Core agencies encompass the minimum functionality required in order to support the agents. A core agency provides the following services:

- **Registration Service.** The registration service of each agency maintains information about all the agents hosted by the agency. This information is used for the agent management and the communication between agents. Furthermore, the registration service of each agency is connected to the region registry, which maintains information about all the agents, places and agencies inside the region. This enables the Agency Domain Service (ADS) of the region to provide "white pages" services to agents and communication proxies.
- **Communication Service.** This service is responsible for all remote interactions that take place in Grasshopper, such as location-transparent inter-agent communication, agent transport and localisation of agent through the region registry. The underlying communication protocol can be IIOP, Java RMI or plain socket connections. The latter two can be secured by use of the Secure Socket Layer (SSL). The communication service supports four types of communication: synchronous, asynchronous, multicast and dynamic method invocation.

- **Management Service.** This service enables the human operators to monitor and control entities within an agency (agents and places). It provides the functionality to:
  - Create, remove, suspend and resume agents, services and places.
  - Access information concerning specific agents and services.
  - View lists of the hosted agents and places.
  - Configure the agency by specifying system, trace, security and communication properties.
  - This functionality can be accessed either through the user interface, or through the API.
- **Transport Service.** It supports the migration of agents between agencies. Migrant agents continue their execution exactly at the point they left it before the migration. The transport service handles the import-export of agents, as well as the co-ordination of the actual transfer, which is performed by the communication service.
- **Security Service.** There are two discrete security mechanisms in Grasshopper: internal and external security. Internal security encompasses the protection of the agency's resources from unauthorised use by the agents, as well as agents from each other. It is based on the authentication of the user that executes an agent. The agent effectively inherits the authorisation level of the user that executes it. External security protects the remote interactions between agencies and region registries. It is based on X.509 certificates and the SSL protocol.
- **Persistence Service.** It allows the storage of agents and places in the local file system, so that they can be resumed even after the agency has been shut down and restarted. There are two different persistence services:
  - The implicit persistence service is controlled by the agency administrator. When it is activated, places created within the agency are stored persistently. Hence they are recovered after a shutdown-and-restart. There is also the option to store the state of all the agents when the agency shuts down.
  - The explicit persistence service gives the agent programmers several options: it allows the storage of agents either once or periodically, in intervals specified by themselves. It also supports agent termination, after a specified amount of idle time. The agents that are terminated thus, can be restarted at any time when another entity tries to access them.

3.5.3.3 Agent Communication

Agent communication in Grasshopper is based on classical remote procedure call (RPC). The only innovation of Grasshopper in this area is the provision of a location-independent method to call remote procedures. That is, the steps required to call a procedure of another agent are unique, whether the agent resides on the same, or on a different platform. Communication is performed following the client-server model. The client and server roles are pertinent to a single communication session only, and roles may freely be swapped for subsequent communication. Thus communication in general is symmetric (an agent can be both a client and a server), but communication sessions are asymmetric.

Nevertheless, the Grasshopper API can easily be extended to include the functionality required to support message-based communication. That is exactly what the Grasshopper ACL add-on does, providing full FIPA-ACL compatibility. However, one must bear in mind that this not a built-in feature of the platform.

The communication service is responsible for both agent migration and agent interaction, either local (within the agency) or remote. When using the communication service, clients do not have direct references to the corresponding servers. Instead, an intermediate entity is introduced, named proxy object or simply proxy. In order to establish a communication connection with a server, a client creates a proxy that corresponds to the desired server. This proxy in turn establishes the connection to the actual server. In the context of Grasshopper, three different kinds of servers exist: agencies, agents, and agency domain services (regions). The following figure visualises this model of communication.

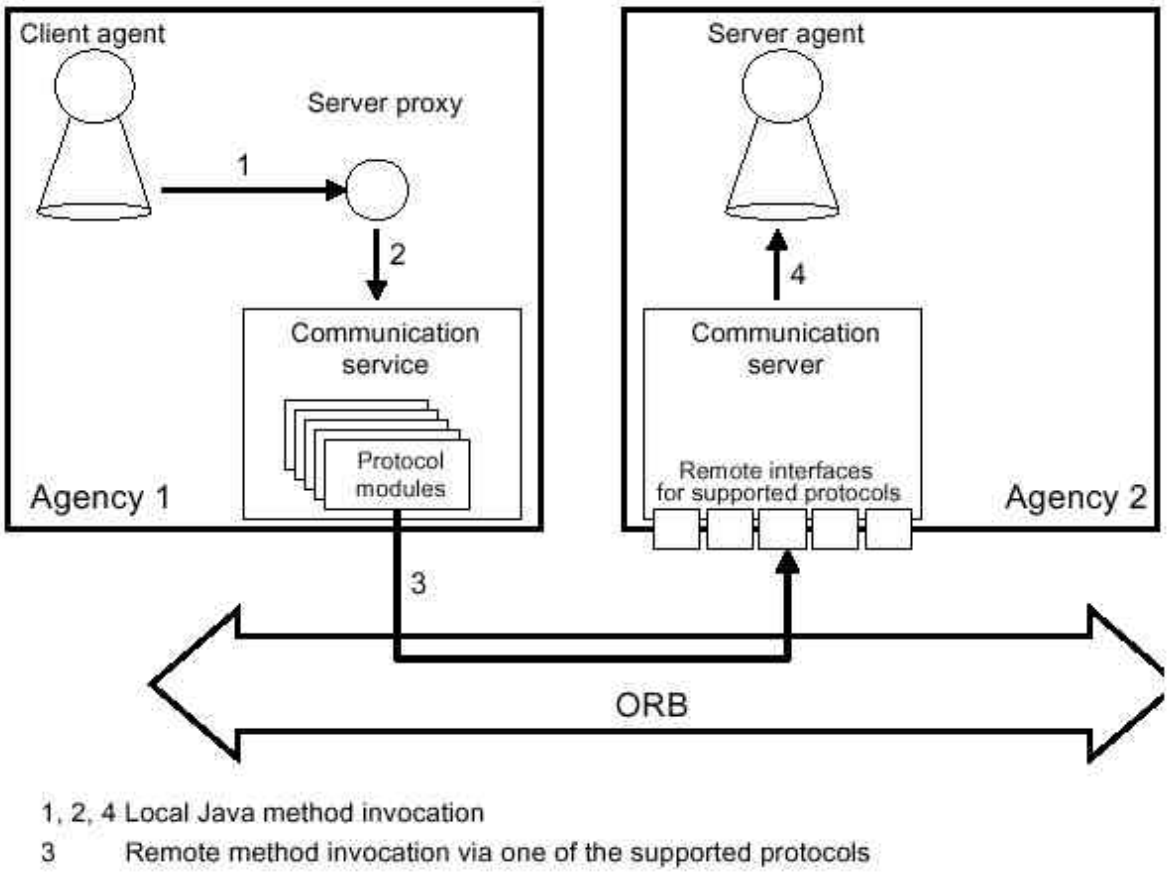

Figure 3.12: The Grasshopper communication model

If an agency domain service is running and both the client's and the server's agencies are registered at this service, a location-transparent communication session can be established. The client simply provides the identifier of the demanded server, and the proxy object automatically contacts the agency domain service in order to determine the server's location. If the server is realised as a mobile agent that moves to another location, the proxy at the client side keeps track of the server by requesting its new location from the agency domain service.

In practice, the server side of a Grasshopper communication connection is realised in terms of a Java object that provides at least one public method to the communication service. All methods that are to be accessible via the communication service have to be included in a Java interface that is implemented by the server object. The interface must be known to the client at compile time, in order for it to be able to access the server's methods.

Communication can be either synchronous or asynchronous. Synchronous communication means that, after invoking a method of a server agent via its proxy, the client agent is blocked until the invoked method returns. In contrast to this, an asynchronously invoked method does not block the invoking client. That means, the client may continue its task after invoking the method, while the corresponding server performs the invoked method in parallel. This does not cause any problems for the client if the invoked server method neither returns a result nor throws an exception. However, if a result or an exception is to be transmitted by the server, the client must have the capability of receiving it.

For this purpose, a proxy that supports asynchronous communication implements the interface de.ikv.grasshopper.communication.IFu-tureResult, which provides a method named getFutureResult(). This method, invoked by the client, returns an instance of the class de.ikv.grasshopper.communication.FutureResult, which represents an intermediate storage for asynchronously arriving results. That means, when an asynchronously invoked method returns a result in terms of a return value or an exception, this result is transmitted to and maintained by the FutureResult object of the server proxy. The client may perform its own task in parallel without being influenced by the interactions between the server and its proxy. The FutureResult object offers methods that implement the functionality required to retrieve asynchronously arriving results and exceptions check for timeouts etc.

3.5.3.4 Summary

Grasshopper, currently in its second version, is a mature agent platform implementation. It is accompanied by fairly good documentation and examples, giving the unfamiliar programmer a head start. Its deployment is easy and, being implemented in Java, it is portable to most popular computing platforms. Its core strong point is the conformity with the MASIF standard. It can be enhanced with modern security add-ons.

Nevertheless, Grasshopper is a representative of the old-fashioned agent communication school, being essentially based on RPC. Grasshopper's communication method is a sophisticated extension of RPC, in the sense that it supports location-transparent and asynchronous communication. Still, the communication between two agents does not get over the RPC paradigm, in that it requires that the client knows a priori the semantics of the process (the method's signature, in Java terminology) it wants to call.

This low-level communication principle, though effective in many applications, is considered obsolete and unsuitable for intelligent agent and multi-agent systems. Grasshopper's answer to this is the FIPA ACL compatibility add-on. However, at the time of writing of the report at hand, the Grasshopper ACL add-on was available from neither IKV++ nor GMD FOKUS, and no relevant information was provided. This is a severe potential drawback, given the imminence of MKBEEM's implementation. A second, equally important drawback is the quality of the accompanying documentation. Both the user's manual [32] and the programmer's manual [31] were incomplete, having whole chapters missing.

3.6 Platform – Independent Approaches

The most common approach to building distributed agent systems is the use of custom platforms, based on some form of remote procedure call (e.g. RMI, CORBA), instead of using a third-party platform. Even if the agent paradigm is adopted, an off-the-shelf agent platform is not a mandatory choice, since it is quite possible to implement the required functionality using only CORBA or RMI. For the purposes of the MKBEEM project, it is particularly important to examine the approaches followed by two recent EC-funded projects, ABROSE and ABS, which have followed this strategy. MKBEEM can easily inherit know-how from these, since members of the MKBEEM team took part in them. We review these projects here, emphasising on results that MKBEEM can draw on.

3.6.1 ABROSE

ABROSE is a framework for an Electronic Brokerage Service based on the emerging agent technology. Electronic Brokerage Services' main objective is to help users/customers find the offer closest to their demands, as well as to enable providers present their services/products in public [41].

The main objectives of ABROSE were to design, implement and integrate an agent based brokerage system for e-commerce, based on the achievements and results of the AC 206 ABS project. This system incorporates the following features:

- *Matching of demand and supply:* improve the performance and the quality of mediation through usage of a collaborative adaptive society of agents.
- *Remembering success and quality of past transactions (collective memory)*: for the benefit of the end users, content providers and market place service operator.
- *Propagating offers and demands*: by using accumulative knowledge about end-users and content providers.

3.6.1.1 Functionality

ABROSE meets the needs of users and content providers to reach each other by appropriately matching the user's demands and the content provider's offer. Some of the issues and challenges that ABROSE had to deal with are:

- Creating and maintaining a structured internal representation of the heterogeneous (both in access and used data formats) dynamic provider domains.
- Enabling users to view and access the provider's domains and offers, and at the same time enable content providers to propagate their offers towards end users, by an offer registration mechanism.
- Knowledge capture and representation:

- capture and representation of subjects, concepts and relations between them, for the representation of the provider domain treated by the broker
- capture and representation of content providers and their coverage of the subjects and concepts

- Request Processing. The main steps are:
  - To identify the appropriate subjects and concepts and the content provider, related to the query
  - To access the content providers
  - To retrieve the actual offers, prepare, compute and present the results in an appropriate manner
- Offer Processing/Propagation. Using the knowledge of the provider domain, the system has to:
  - enable content providers to register or update their offers efficiently
  - associate the offer with the brokers' subjects and concepts

3.6.1.2 ABROSE Multi-Agent Architecture

In general, a multi-agent system (MAS) is viewed as a system composed of many interacting agents. The technical approaches to implement individual agents or a society of agents can be similar: each can have an individual goal. The main difference is that the multi-agent system also possesses a global task to solve. For example, ABROSE finds efficiently the best relevant matches between offers and demands. The global task is very difficult to solve in a dynamic environment in which unexpected events frequently occur, given the constraint that no agent has a global view of the system. In order to overcome these unexpected events, the basic multi-agent principle is that agent communication is driven by "social conventions". As the global function of this kind of system is not well defined, the consequence is that the classical approach based on a global top-down strategy to design artificial systems is totally inappropriate for Internet. Today no software devoted to access and management of Internet information is able to assume coherence and completeness. Thus, a quite different approach is necessary to guarantee the *functional adequacy*. Functional adequacy is defined as the capability of a system to give at all times the right response expected by its environment. Ideally in the Internet domain, functional adequacy relates to two properties:

- All existing information belonging to a given subject is obtained by an end-user requesting it, and only this.
- Any offer made available by a content provider is broadcast to all the interested end-users, and only to them.

In these systems, the behaviour of an agent in its environment depends on some basic characteristics:

- *Autonomy*. The global system computation is distributed into agents. There is no central management.
- *Partial environment representation*. Any agent knows the state of the world and the activity of others (beliefs). It works within a bounded rationality leading possibly to concurrent situations or conflicts with others.
- *Tasks realisation*. Commonly an agent has not only skills (its competence domain) but also goal(s) to achieve.
- *Communication abilities*. An agent must interact with other agents locally or remotely in using protocols and speech acts.

These agent characteristics are well adapted to the customers and content providers. The dynamics of the marketplace domain is the main reason why ABROSE is oriented towards an adaptive multi-agent system. The global architecture of ABROSE is composed of the *user domain* and the *broker domain*. The user domain consists in all software needed to connect and to communicate with the broker domain and the front-end of the content providers. In the broker domain, the main functions of the brokerage service take place. Because ABROSE views information brokerage as a symmetrical process, giving users and content providers the opportunity to reach each other, the user domain concerns a customer as well as a content provider. So the term "user" represents indifferently a customer or a content provider.

Figure 3.13 shows the Global Architecture of ABROSE. In the rest of this section explanation of the abbreviations and of the main tasks of each functional block shall be provided.

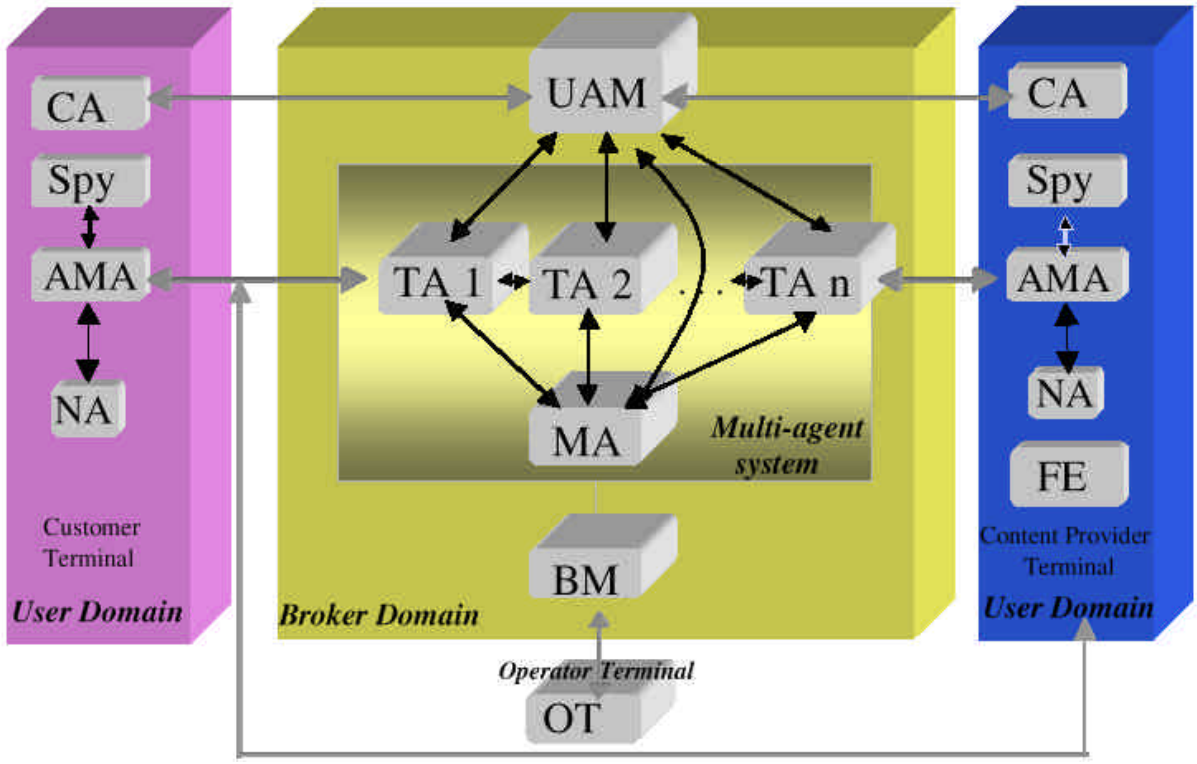

Figure 3.13: ABROSE Global Architecture

The User Domain

- **Connection Assistant** (CA): it helps the user to connect and to register to the system.
- **Agent Manager Assistant** (AMA): it co-ordinates the agent-based communications between the user and the broker domain. The Graphical User Interface belongs to it.
- **Navigation Assistant** (NA): it helps the user to navigate in the broker knowledge space and select the relevant domains/criteria for his/her request.
- **Spy:** it takes the content of a transaction done by the customer to the content provider in order to improve their reciprocal user profiles.
- **Front End** (FE): it is used for the mapping of agent requests into specific content provider database capabilities.

The Operator Terminal

The OT helps the manager operator to access the ABROSE system in order to manage and administrate the service provided by ABROSE. It displays the status of the overall blocks of the system (visualisation), facilitates the access to parameters provided by each component and permits access to information related to all the users. This block is created when the manager operator is interested in performing any operation (visualisation, service monitoring, network and infrastructure monitoring, and profile management).

The Broker Domain

It has the following three sets of components:

- **The User Access and Authentication Manager** (UAM) controls the access to ABROSE by using a login and password. This block is created during the launch of the broker system and must be running while the broker service is available.

- **The Broker Manager** (BM) monitors the interactions between all functional blocks. IBM has provided the visualisation and monitoring of the system. This block is also created during the launch of the broker system.
- **The MAS (Multi Agent System).** Each user has its own associated agent called “Transaction Agent” (TA). A TA co-ordinates the communications between the user it is associated with and the broker domain. TAs are grouped together under the control of a “Mediation Agent” (MA). The MA controls the creation of TAs associated with new users and with the removal of TAs associated with dismissed users.

There are three types of co-operative agents in ABROSE: the *Mediation Agents*, the *Transaction Agents* and the *Beliefs Agents* following the same "social conventions" (Figure 3.14). An agent at a level is composed of co-operating agents at the sub-level.

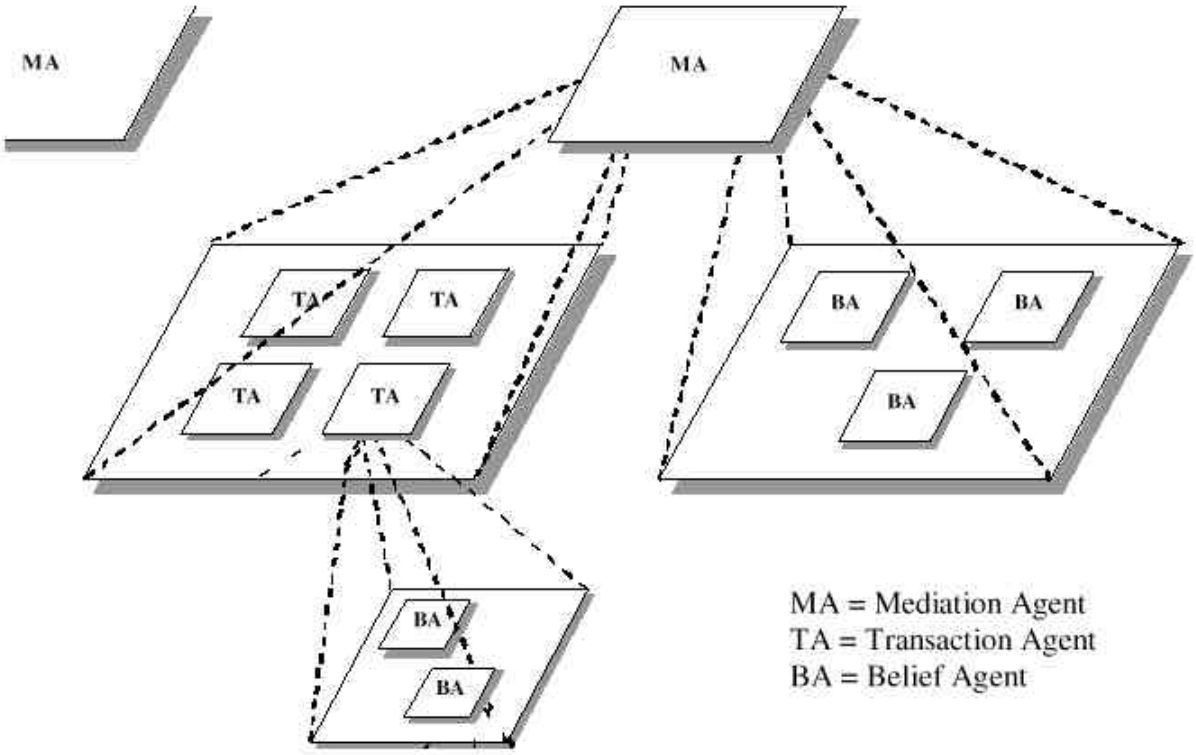

Figure 3.14: The three-layered Multi-agent Architecture

The top layer is composed of Mediation Agents (MA) distributed on interrelated computers. MAs can be added or suppressed inside the system, thus avoiding the need to maintain any global view. In their second role, these agents could also be associated with service or user domains such as the set of users registered in the same Internet Service Provider or a collection of services sharing the same topic. Their goal is to structure the set of users and services in order to decrease the complexity of locating them. The main difference of an MA and a “yellow page” service is in fact that an MA learns from the transactions done in the system and it does not require an update by the system designer. *The Mediation Agents play the same role as the Directory Facilitator in FIPA ACL 97 for electronic commerce*.

The second layer is made of Transaction Agents (TAs), which represent Customers and Content Providers. Content Providers provide services in ABROSE and Customer are users who would like to find an agent that provides a particular service. They are built when a new service is created or when a customer registers. TAs have beliefs about the skills of themselves and of others TAs, which are hosted by the same MA. The TA is in charge of autonomously exchanging requests or answers with other TAs. The TA should also communicate with the customer or content provider it is associated with.

All agents of the two previous types possess beliefs about themselves and other agents belonging to the same level. These beliefs describe the organisation of the agent society. They constitute the third layer of multi-agent system in ABROSE called the Belief Network (BN) that is composed of Belief Agents (BA). A BA expresses the viewpoint that an agent (MA or TA) can have about itself or another

agent. A BA links the common concepts of two agents and confers on an agent the capability to reason about others. It gives the possible interaction links between the agents of the society.

3.6.1.3 Technologies and Tools Involved

The ABROSE platform V2 is implemented in Java with JWS1.1.3, the communication between the Broker and the User domains is developed with OrbixWeb 3.1 (a Java implementation of CORBA from IONA). The brokers run under Solaris2.6. All the specifications have been made in UML under Rational™ Rose98™. The client is a standard PC equipped with a standard browser. There are several versions of Netscape suitable for this purpose: from 4.05 to 4.61 and up. They can be used both in Windows (95, 98 or NT) or Unix (Solaris, UX or Linux) environments. The package `swingall.jar` version 1.1 must also be added in the classpath.

FIPA-ACL

Each agent of ABROSE is endowed with a co-operative social attitude. This implies three main properties:

1. Sincerity. All information sent is assumed to be true from the viewpoint of the sender
2. Willingness. All agents try to satisfy a received request if it is coherent with its own skills and the current state of the world
3. Reciprocity. Each agent knows that all the others are co-operative too.

These requirements lead to some consequences:

1. A co-operative agent cannot act without some beliefs about others.
2. Good intentions. An agent could send spontaneously information, without any previous demand, if it believes in its usefulness for the receiver.
3. If an agent is unable to satisfy a given request, it recruits automatically others agents that it knows a relevance on the subject.

In ABROSE, agents interact by way of messages that encode the communicative acts.

The Brokering and recruiting request protocol is composed by the following ordered list of acts.

1. **Inform.** This basic act is performed if the agent believes that the object corresponding to the definite descriptor is the one that is given, and does not believe that the recipient of the act already knows this [5].
2. **Request *.** The agent is unable to perform the previous inform, but it believes that other agents are able to do it. The request is exactly the initial request it receives sent to the set of believed relevant agents. This process is called relaxation in Abrose and recruiting in [42]. In Abrose, relaxation is done by Transaction Agents or by Mediation Agent.
3. **Request **.** The agent is unable to perform inform and also unaware of other relevant agents to perform it. In this case it creates a new request sent to another agent on the upper level such as directory facilitator. The response is returned to it, and is in charge of sending back it to the initial agent. This corresponds to a brokering activity.
4. **Not-understood.** In the other cases

The Brokering and recruiting query protocol is composed by the following ordered list of acts.

1. **Inform-ref**. This basic act is performed if the agent believes that the object corresponding to the definite descriptor is the one that is given, and does not believe that the recipient of the act already knows this.
2. **Query-ref *.** The agent is unable to perform the previous inform-ref, but it believes that other agents are able to do it. The query-ref is exactly the initial query-ref it receives sent to the set of believed relevant agents. This process is called relaxation in Abrose.
3. **Query-ref **.** The agent is unable to perform inform-ref and also unaware of other relevant agents to perform it. In this case it creates a new query-ref directed to another agent on the upper level such as directory facilitator.
4. **Failure**. A co-operative agent returns a failure only if all the previous acts cannot be performed.

5. **Not-understood**. In the other cases

The goal of the twenty communicative acts defined in ACL is to assume completeness, simplicity and conciseness without any presupposition on the social behaviour of the agents. But in the case of co-operative agents such as in ABROSE all these acts can be simplified. For example:

- When a co-operative receiver agrees with the message content, no returned acknowledge is necessary. That suppresses the communicative acts 'accept-proposal' and 'agree'. This first point implies that a great number of communicative acts are suppressed when agents are supposed to be co-operative: the sender deduces implicitly that no response corresponds to acceptance of the information content by the receiver.
- 'Propose', 'call-for-proposal', 'accept-proposal' and 'reject-proposal' are not useful for a co-operative agent who tries to satisfy the others all the time. All communicative acts are implicitly considered as a proposal, and a negotiation process between agents can be observed in using only the previous defined basic acts.
- 'Request-when', 'request-whenever', 'subscribe' are acts specifying interest of the sender about some information or performing some action. A co-operative agent, who believes that another agent is interested, acts automatically for its satisfaction.

Ontologies

When two agents wish to converse, they must share a common ontology for the domain of discourse, in order to avoid misinterpretations of message contents. In the general case of human speech acts the ontology is dynamically constructed during the interaction process by mutual adjustment.

The specifications of FIPA97 ACL make no provision for dynamic sharing or updating of ontologies[3]. This is the reason why a message structure of ACL includes a parameter called "ontology". Thus, an agent is responsible for ensuring that it uses the same ontology as the sender of a message. A basic capability of all ABROSE agents is learning about the skills of others: this is necessary in the brokerage domain where information objects are highly dynamic. When a given transaction between two agents ends, the sender and the receiver update their respective previous description. This explains how a message structure in ABROSE does not contain an ontology field: it is supposed that the learning process gives to all the agents the ability to interact in a relevant domain of discourse. If this is not the case at a time, the learning process will correct the ontology description for further transactions.

3.6.1.4 Exploitable Results

Having provided an overview of the ABROSE project, we discuss in this paragraph the significance of the project's achievements, from MKBEEM's point of view. For this purpose, we can distinguish three main areas:

- The Agent Platform
- The Communication Language
- The Architecture

The Agent Platform

ABROSE aimed at the exploitation and expansion of the ACTS ABS project. Improvements were made in the performance of mediation through the usage of a collaborative adaptive society of agents. Furthermore the matching between demand and supply became more accurate by remembering the success and quality of past transactions. ABROSE used mobile agents that were able to intercommunicate in their effort to extend their knowledge of the 'world' in which they existed. The agent platform of ABROSE was designed and developed by the University of Toulouse. The tool behind the agents' communication is a Java implementation of CORBA, OrbixWeb 3.1. Experience showed that CORBA is not a good solution as far as the user agent is concerned, as the client will be forced to download OrbixWeb Java classes, a burden that should be avoided. However, this issue is not applicable if the user agent is configured to run on the server, rather than on the client.

The ABROSE platform seems attractive if we decide on keeping a history of transactions (success/failure) in order to improve the mediation. In such a case though, a possible alternative would

[3] Currently this issue has been addressed by FIPA, at least partially, with Part 12 of the FIPA98 specification, titled "Ontology Service".

be to build an ontology (or some other structure) on which this knowledge can be mapped and updated dynamically.

The Communication Language

The agent language used in ABROSE is actually a subset of FIPA's ACL. The 'message-types' that were not implemented were not necessary, due to the special social behaviours of the agents and the internal conventions that were implied. More details can be found in *FIPA-ACL* section 3.5.1.4 [5]. This approach may very well be adopted by MKBEEM. With the addition or modification of some communicative acts the agents in MKBEEM will be able to co-operate and complete their tasks adequately.

The Architecture

The architecture of ABROSE may seem at first very similar to that of MKBEEM. The experience from the design and evaluation of ABROSE's architecture can help us on a more practical and clear construction of our own functional blocks. Certain features of MKBEEM though (like multilinguality, extensive use of ontologies, etc) will play a decisive role in our choices.

Despite the apparent similarity, the two projects follow quite different architectural approaches. ABROSE relies on a society of co-operative agents, populated by instances of identical agents. In MKBEEM the agent world is very sparsely populated and co-operation takes place only between different agents (i.e. the Rational Agent is expected to co-operate with the User Agent, but not with another Rational Agent). Moreover, MKBEEM pursues a new goal, not addressed in ABROSE: the natural language interface. In ABROSE, intelligence was required for the agents to interact among themselves, whereas in MKBEEM it is required to interact with the user. Undoubtedly it is too soon to decide on architecture at this early point, but the similarities and differences of the two projects have to be taken into account, in order to capitalise effectively the experience from ABROSE during the design and implementation phases.

3.6.2 ABS

3.6.2.1 Overview

The ACTS ABS (***Architecture for Information Brokerage Service***) AC 206 project, previously referenced, has been focused on the design, specification, implementation and validation of an open broker architecture to permit the efficient provision of online information services, in the context of electronic commerce, over the forthcoming European Information Infrastructure. The project main objectives were:

- To define and specify Information Brokerage Service Architecture based on ODP-RM and TINA concepts.
- To design and implement prototypes of an open brokerage system using CORBA and Java.
- To validate the service by conducting regional and international trials in real life electronic commerce environments.

The projects specification and implementation approach has been based on ODP-RM concepts, as mentioned above, and UML. The Service and its environment have been also specified according to the ODP-RM viewpoints, which have been used to provide different models:

- Enterprise model: defining the service from the viewpoint of the organisations and people, which will use and operate the service, and their interactions.
- Information model: defining the information, the associated semantics, and the information processing activities of the system.
- Computational model: describing the application in terms of computational objects and to define the rules how these objects interact with another.

The specification and implementation of the Computational objects has been performed using the UML methodology, as well as parts of the enterprise modelling (service definition).

3.6.2.2 Functionality

As a first result a comprehensive enterprise model was developed. Brokerage is viewed as a service component of a general mediation platform. Its main task is to supply the symmetrical conjunction of demand and supply processes. In order to facilitate this, the following necessary functionalities (see Figure 3.15) were identified:

- Demand definition support.
- Catalogues management.
- Directory and navigation functions.
- Offer registration.
- Federation function.
- Dynamic search.
- Interfaces to external functions.

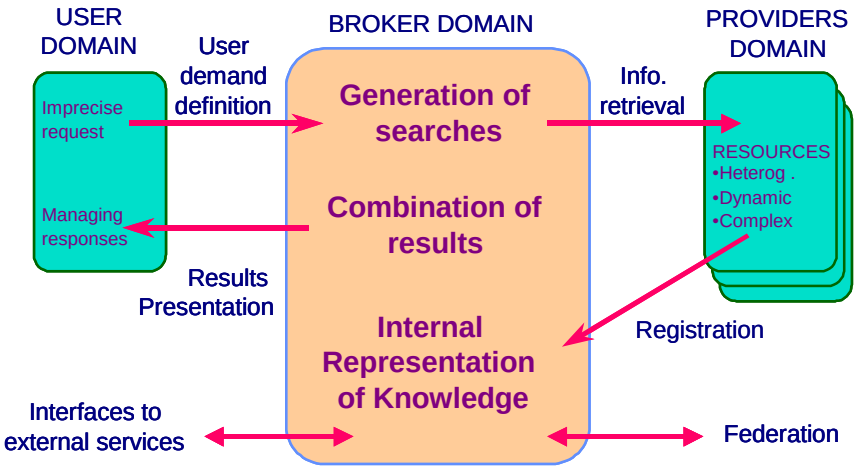

Figure 3.15: ABS Functional Interfaces

Interfaces with external components provide supporting capabilities: like authentication, billing and accounting.

3.6.2.3 Architecture

To have an idea of how a brokerage platform could be implemented, the ABS architecture definition is going to be commented based on the previous enterprise model. The different components will be described in order to analyse how have the different brokerage tasks been distributed.

Figure 3.16 shows the ABS Brokerage service functional blocks derived from the application of the ODP computational viewpoint. In it, some different domains can be identified:

- Broker domain: that in which the activities of the *Brokerage Service Provider* are performed
- User domain: that with which the *Customer* accesses the *Brokerage Service*.
- Provider domain: that with which the *Provider* accesses the *Brokerage Service*.
- External broker domain: the architecture allows the federation of different brokers.
- Operator domain: that with which an operator can manage the *Brokerage Service*.

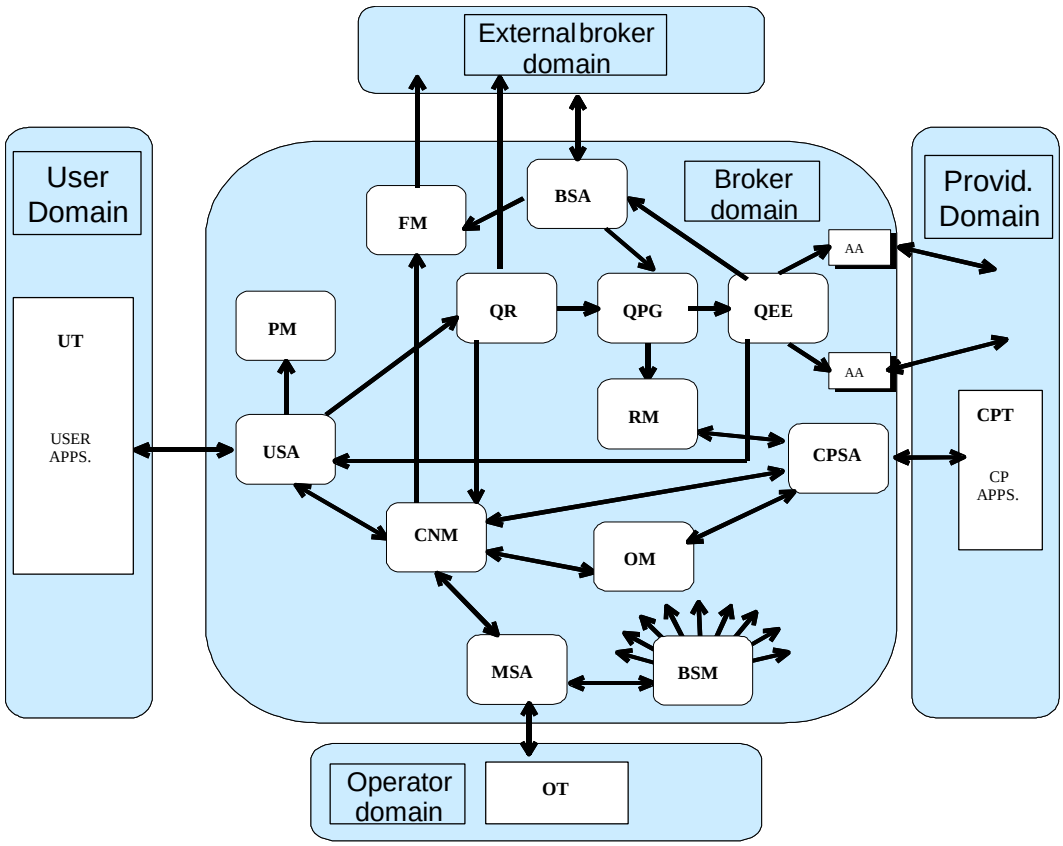

Figure 3.16: Blocks of the Broker System.

From a computational viewpoint, a set of functional blocks, from the Broker, User, Provider and External Broker domains, can be grouped in what could be called *Access System*. With it, the service components are created by the access components after a successful access session

The *Access System*, depicted in figure 3.17, has been defined following TINA service architecture principles as close as possible. In this way, the ABS brokerage service might eventually be included in a TINA service environment in an easy way.

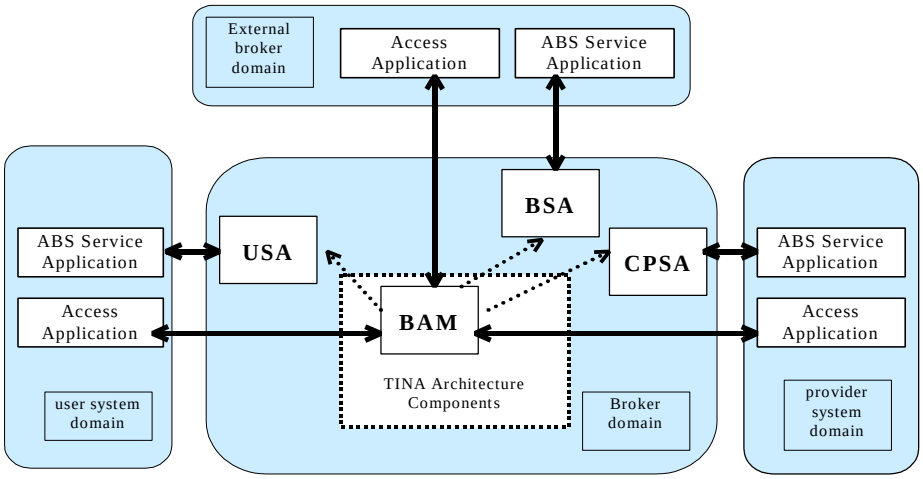

Figure 3.17: Blocks of the Access System.

The following list shows the function of each component of the figures above. All them have been implemented as CORBA objects.

- **AA** (*Access Agent*): they are responsible for information retrieval in the Content Provider domain. As each provider has its own resources there is one Access Agent for each provider, to adapt those resources to the broker domain.
- **BAM** (*Broker Access Manager*): it completes the access sessions of the ABS system.

- **BSA** (*Broker Service Agent*): it is the broker's domain end-point of a service session with an external broker.
- **BSM** (*Broker Service Manager*): it controls the interactions between all other functional blocks and related components.
- **CNM** (*Conceptual Network Manager*): it is responsible for managing the Conceptual Network objects (broker knowledge).
- **CPSA** (*Content Provider Service Agent*): it is the broker's domain end-point of a service session with a content provider.
- **CPT** (*Content Provider Terminal*): the content provider uses it to access the broker system and to request the broker services.
- **FM** (*Federation Manager*): it is responsible for managing all the viewpoints exporting/importing activities related with the federation functionality.
- **MSA** (*Management Service Agent*): it is the broker's contact point for the broker operator.
- **OM** (*Offer Manager*): it manages the Content Provider offers
- **OT** (*Operator Terminal*): the broker operator uses it to access the broker system and to manage and administer the broker system
- **PM** (*Profile Manager*): it is responsible for managing all the actor's profiles in order to use them for completing in an appropriate way the requested service
- **QEE** (*Query Execution Engine*): it analyses the plan representing the query and call specific AA to execute different subgoals in the Content Provider domains
- **QPG** (*Query Plan Generator*): the QPG gathers all the needed information from other components in order to build semantically correct and executable query plans for answering user's queries.
- **QR** (*Query Redirector*): it is responsible for redirect a user query to the local broker, an external broker(s) or both
- **RM** (*Resource Manager*): it manages the Content Provider resources descriptions
- **USA** (*User Service Agent*): it is the broker's domain end-point of a service session with a user
- **UT** (*Private User Terminal*): it is used by Private Users to access the broker system and to request broker services

3.6.2.4 Information Model

The information model, obtained from the application of ODP information viewpoint focuses on semantic aspects of the service, trying to describe what information is needed to support the brokerage tasks and how that information is processed. Both the information processing, the information retrieval and the directory functions are required functionalities devoted to the added values support offered by the ABS brokerage service architecture.

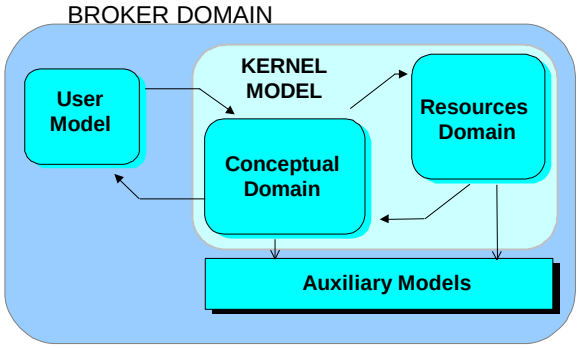

Figure 3.18. ABS Information Sub-models and their relationship.

The information model must reflect all the aspects related to the handling of information associated to those functionalities. Using this constraint, the information model has been divided in several interrelated submodels as shown in figure 3.18:

- *User information model*: it contains information about the users of the service. This information is intended to facilitate the query and navigation processes and to customise the service for each user. The key aspect of this submodel is the *user profile*. Attitudes, skills, preferences, etc. of each user are the typical information to be included in its *user profile*.
- *Conceptual domain*: this submodel contains information about the different business areas that are being supported by the brokerage service. The main aspect of this conceptual domain is the so-called *Conceptual Network*. It can be defined as a complex multirelational structure of nodes and relations among nodes. Each node represents a concept of the real world, and a set of concepts, and their relations, can be projected over the so-called *viewpoint*, which constitutes an abstract representation of a particular business area. Figure 3.19 shows the ABS vision of the *Conceptual Network*. In that figure we can appreciate the *Conceptual Network* (CN) on the top. The concepts of the CN are grouped according to different business areas. Each group of concepts (each business area) constitutes a *viewpoint*, which is composed of a set of *Shadow Objects* (SO). A SO could then be defined as the projection of a concept over a particular *Viewpoint*. The same concept from the CN could be projected over different *Viewpoints* thus creating different SOs. SOs derived from the same concept are related by the so-called *Bridges*.

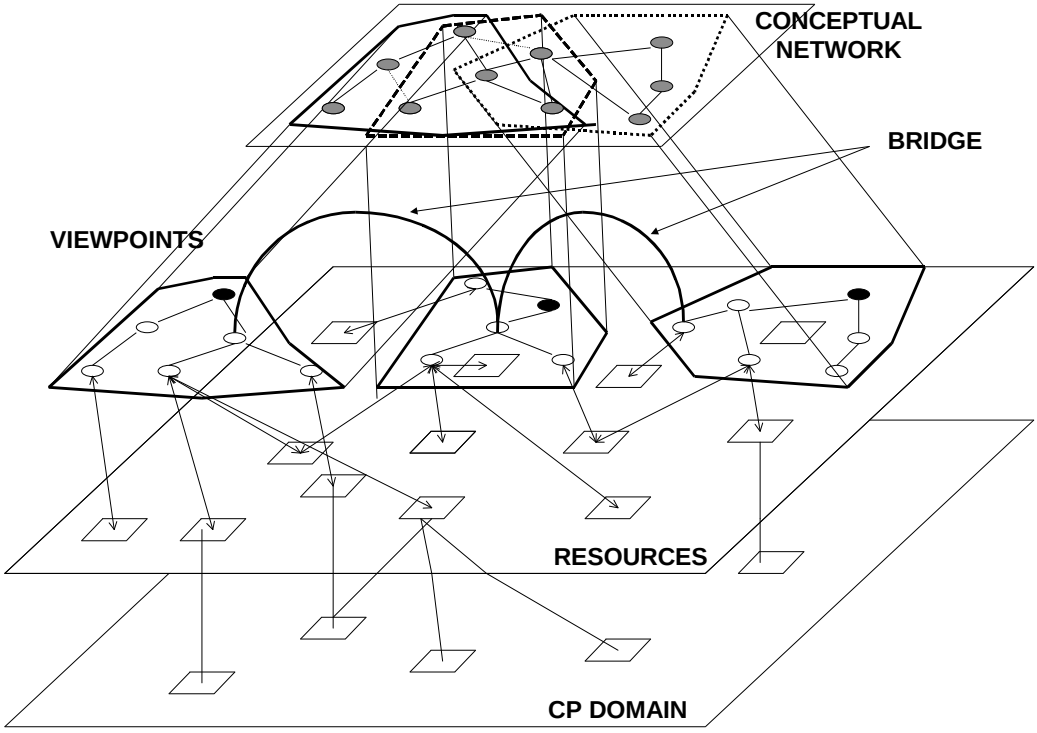

Figure 3.19: The ABS Conceptual Network.

The way the *Conceptual Network* has been designed has two very important advantages:

1. *It is a general model*: it is very easy to add new business areas. The ABS electronic broker might be able to cope with almost any business area.
2. *It is a flexible model*: by means of the *viewpoints*, the user can navigate through different business domains. Furthermore, by means of the *bridges*, the user is able to go from one business area to another very easily.

- *Resource domain***:** a resource is a composed entity that encapsulates all necessary informational items from the providers' domain. It includes *content descriptions* (*what* the providers offer), *query capabilities* (*how* the broker can ask the providers' sites for some needed information), *access descriptions* (indicating how the broker can *access* the providers' domain to make a query), and *Content Provider Profiles* (used for filtering and discrimination during the search process). *Resources* are "attached" to SOs by means of *queries*.
- *Auxiliary Model***:** contains information about external mediation service providers.

3.6.2.5 Technologies and Tools Involved

The ABS Broker prototype took advantage of distributed processing capabilities of CORBA and the flexibility, portability, and object-orientation of Java. For this, OrbixWeb™ 3.0 (the same CORBA implementation used in ABROSE) and JDK 1.1.5 were used. A CASE UML tool (Rational™ Rose™ 98) was used for modelling and IDL interfaces generation. The information contained in the Conceptual Network was stored in Relational DataBases.

The broker, running under Solaris™ 2.5.1, used a full-distributed approach, being each component a CORBA object, as mentioned above. The clients were standard PCs equipped with an Internet browser (at least Netscape™ 4.05) with Java enabled. The communications between the broker and the user domains were using also CORBA, but as it was a particular implementation, the client had to download OrbixWeb Java classes to perform such communications.

As mentioned in 3.6.2.4, the information model used the approach of the so-called Conceptual Network, which was internally mapped into a Relational DataBase.

The Management aspects of ABS were done by using an Operator Terminal based on the Scotty Management platform, running on Solaris 2.5.1. The communications between the manager and the ABS application was based on the SNMP management protocol, with an ABS Management Information Base (MIB) defined specifically for this application. The mapping between the generic ABS SNMP MIB objects and the managed parameters of each CORBA server was implemented by using the JIDM (Joint InterDomain Management) approach, accessing to semi-transparent management instrumentation included in each CORBA object.

3.6.2.6 Exploitable Results

Enterprise Model

One of the key results of ABS was the identification of the enterprise model for the brokerage functionality, and the relationships among the different actors involved in the provision of this service. Several roles for each actor were identified, corresponding to different scenarios of service provisioning. These actors are:

- User of Brokerage service
- Brokerage Service Provider
- Service Provider
- Content Provider
- Retailer
- Network Provider

The detailed specification of the ABS Enterprise Model can be found in the D23 deliverable of the ABS project.

Brokerage Architecture

The brokerage architecture defined in ABS was made in a generic way so that it can be applied to many brokerage scenarios like the one proposed in the MKBEEM project. The architecture was specified from a UML use cases representation of the service requirements, as shown in the next figure.

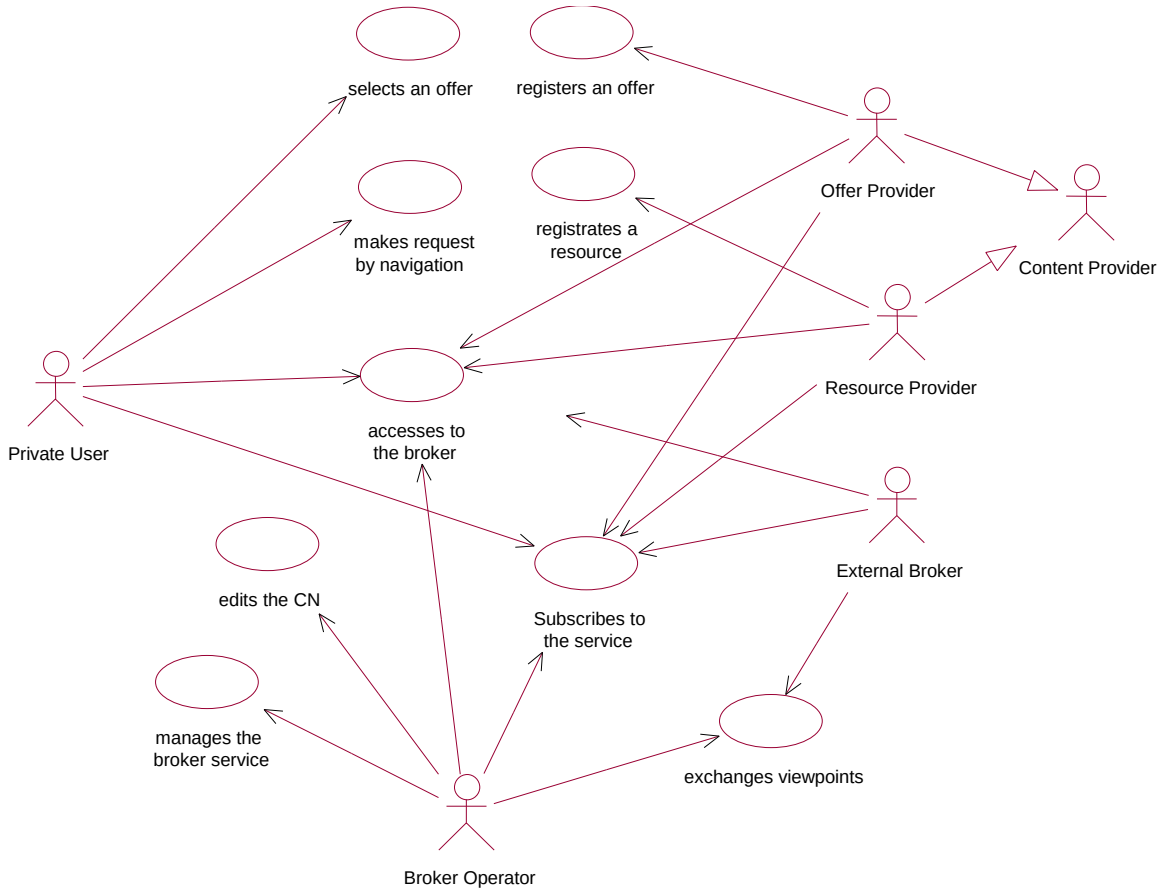

Figure 3.20: The ABS Brokerage Architecture.

In this definition it is embedded the main functionality of a broker:

- For the Content/Service Providers:
  - Register a capability
  - Register an offer
- For the user:
  - Make a query
  - Select an offer
- For an external broker:
  - Exchange knowledge or federate it.
- For the operator:
  - Update the knowledge
  - Manage the service

So, ideas from this architecture can be very useful for its application to MKBEEM, like the ability for a Content Provider of register an offer or register a resource (capability), the federation framework by the interchange of knowledge between brokers, or the use of provider agents for solving the problem of heterogeneity in the provider's domain.

Management Architecture

The management framework of ABS and ABROSE can be also applied to the management and administration of the MKBEEM prototype. In both projects, the management approach used was to identify the manageable parameters of the application in order to instrument the adequate mechanisms for a remote management access to them, which can be integrated in an overall management platform for making an integrated service management.

Although there are some particularisation on the management architecture for accessing the real parameters depending on the chosen architecture (i.e. CORBA for ABS, JAVA-agents and CORBA in ABROSE), the framework and conceptualisation of the Management Information can be reused.

4 Conclusions

As we mentioned in the introduction, the goal of this report is to review and evaluate technologies, platforms and tools that are of potential use to the development of the distributed agent system on which MKBEEM's architecture is based. Having reviewed the state of the art in agent technology and related tools, we now attempt to summarise our findings and identify the main decisions that have to be made, in order to select technologies and tools, and the alternatives for each one.

4.1 Off-the-shelf Agent Platforms vs. Custom Systems

We can divide the approaches to agent systems integration into two separate groups: the first group is based on the employment of classical distributed application programming methods to achieve communication between the agents. On the contrary, the second group is characterised by the existence of an agent platform, often third party provided.

An agent platform can be viewed as a distinct (although presently vague) extra layer in the network protocol stack, residing on top of the transport protocol. Agent platforms provide high-level, purpose-built communication facilities tailored to suit the needs of agent systems. Agent platforms are not application-specific. Rather, they form an infrastructure on which any or most agent systems can be implemented.

It should be clear by this description that the selection of an agent platform is an important commitment for an agent application developer. On the first hand, agent platforms provide valuable functionality off-the-shelf, sparing the effort to implement them using the transport mechanism of the network. They also provide for inter-operability, especially those that tightly adhere to agent standards. They are a good programming concept in general, since they separate the agent application from the logistics of the communication and provide a level of abstraction between the application and the network.

On the other hand, agent platforms are not without drawbacks. Agent platforms are an immature concept and, as such, suffer from a number of problems that mature technologies have grown over. The most severe issue, present in all the platforms we reviewed in varying degrees, is that of robustness. Most agent platforms appeared easy to destabilise on purpose and we experienced a considerable number of crashed and unpredictable behaviour. What is more, the quality of the documentation was average at best. Support or expert advice is not quite easy to get, as a result of the small lifetime of the products and the fact that most of them are developed from small companies or research institutions. Moreover, agent platforms provide much more functions that will be required for the agent system of MKBEEM. While this is an advantage in view of the system's expandability, it comes at the expense of performance.

The platform selection is the first decision point in the technology selection point. Due to the equilibrium of pros and cons, the decision to opt for an off-the-shelf agent platform or build one from scratch is a marginal one and cannot be taken at this early stage. It ultimately depends on two factors, both of which are still fluid:

i. The final user requirements specification (project deliverable D21).

ii. The quality of the available implementations. Although light has been shed on this, time constraints and the number of implementations under review prevent this report from being conclusive on that aspect.

We thus conclude that no commitment to specific technologies can be made at present.

4.2 KQML vs. FIPA ACL

An attempt to further classify agent platforms should take into account their conformity to agent standards. The leading standardisation efforts in the area, as presented in chapter 3, are the FIPA97 specifications, KQML and OMG MASIF. Of those, only KQML and FIPA ACL are mutually exclusive. They are also the most important, from the project's point of view, since OMG MASIF is mainly oriented towards agent mobility. It is therefore useful to classify agent platforms according to the agent communication language they use.

This line of thinking leads us to discover the second decision point in technology selection, that is, which agent communication language we want to use. The quality of the actual products using KQML

or FIPA ACL does not weigh in that decision, since it seems to be equivalent. This fact is to be expected, as FIPA ACL and KQML are very similar, so that the use of one or the other will neither benefit nor handicap a product. In any case, JATLite and JKQML appear to be of the same quality with ZEUS and JADE. It should be noted here that pricing and licensing is not an issue here. These products are provided OpenSource without charge.

The specifications themselves do not have major differences. FIPA ACL is newer and its resemblance with KQML is not coincidental: in fact, it has been built based on the experience of KQML. Indeed, FIPA ACL is generally believed by the scientific community to be slightly better. More importantly, FIPA ACL is more widely adopted, at least in Europe and Japan. FIPA itself is more prestigious than the ARPA Knowledge Sharing Effort, being independent, international and active in all aspects of agent systems, rather than just knowledge sharing and communication.

As a result, we conclude that the facts are in favour of FIPA ACL. Therefore, if the decision is taken to utilise an off-the-shelf agent platform, it should be FIPA ACL compliant. This gives us three choices: JADE, ZEUS and Grasshopper. While we cannot distinguish the first two in terms of quality, it is quite clear that Grasshopper falls short of them. The reasons for this are purely practical and comprise the unavailability of the Grasshopper ACL add-on, and the problems with Grasshopper's documentation. The same applies to FIPA OS, although less strictly. Its documentation is rather poor, and it is not particularly friendly to the inexperienced agent developer. This leaves JADE and ZEUS as the two best choices.

4.3 Summary and Future Work

The selection of the technology for the MKBEEM agent platform can be broken down in three decisions:

- Utilisation of an off-the-shelf product or building of a custom platform based on classical, well-known RPC methods. This decision remains open, depending on the final user requirements specification and the quality of the available tools.
- In the case the first choice is made, commitment to an agent communication standard. We concluded that the best choice here would be FIPA ACL.
- Selection of a product, compliant to the standard we opted for. The choices here are mainly JADE and ZEUS, while FIPA OS and Grasshopper appear to be the outsiders.

We will continue working for the technology selection focusing on two tasks. The first is the translation of the user requirements into functional requirements and matching of these against the alternatives. The second comprises further experimentation with the platforms identified as the having the greater potential, in order to determine their quality and suitability for MKBEEM. The results of these two activities, combined with the insights offered by the report at hand, will allow the rational choice of technologies to best suit the MKBEEM project.

Bibliography and References

[1] *MKBEEM ANNEX 1*
PROJECT REFERENCE IST-1999-10589

[2] *MKBEEM CONSORTIUM AGREEMENT*
REF DELIVERABLE D12,MARCH 2000

[3] *MKBEEM CONTRACT PREPARATION FORMS*
PROJECT REFERENCE IST-1999-10589

[4] *VIVID AGENTS - HOW THEY DELIBERATE, HOW THEY REACT, HOW THEY ARE VERIFIED. EXTENDED VERSION OF G. WAGNER: A LOGICAL AND OPERATIONAL MODEL OF SCALABLE KNOWLEDGE- AND PERCEPTION-BASED AGENTS, IN W. VAN DE VELDE AND J.W. PERRAM (EDS.), AGENTS BREAKING AWAY, PROC. OF MAAMAW'96, SPRINGER LECTURE NOTES IN ARTIFICIAL INTELLIGENCE 1038, 1996.*

[5] *AGENT COMMUNICATION LANGUAGE, FIPA 97 SPECIFICATION. AVAILABLE AT HTTP://WWW.FIPA.ORG*

[6] *G. WAGNER, FOUNDATIONS OF KNOWLEDGE SYSTEMS WITH APPLICATIONS TO DATABASES AND AGENTS. KLUWER ACADEMIC PUBLISHERS, 1998.*

[7] *THE HOMEPAGE OF JADE: HTTP://SHARON.CSELT.IT/PROJECTS/JADE/*

[8] *THE HOMEPAGE OF JESS : HTTP://HERZBERG.CA.SANDIA.GOV/JESS/*

[9] *K. TAVETER, G. WAGNER, COMBINING AOR DIAGRAMS AND ROSS BUSINESS RULES' DIAGRAMS FOR ENTERPRISE MODELING, ACCEPTED TO THE SECOND INTERNATIONAL BI-CONFERENCE WORKSHOP ON AGENT-ORIENTED INFORMATION SYSTEMS (AOIS-2000), 5-6 JUNE 2000, STOCKHOLM (SWEDEN) AND 30 JULY 2000, AUSTIN (TEXAS, USA).*

[10] *V. VASUDEVAN, COMPARISON OF ACLS. AVAILABLE FROM HTTP://WWW.OBJS.COM/AGILITY/*

[11] *M. GENESERETH & R. FIKES, KNOWLEDGE INTERCHANGE FORMAT, VERSION 3.0. REFERENCE MANUAL, COMPUTER SCIENCE DEPARTMENT, STANFORD UNIVERSITY. AVAILABLE AT HTTP://WWW-KSL.STANFORD.EDU/KNOWLEDGE-SHARING/PAPERS/README.HTML#KIF*

[12] *T. FININ & Y. LABROU, KQML AS AN AGENT COMMUNICATION LANGUAGE, IN SOFTWARE AGENTS. BRADSHAW, J.M. (ED.), MIT PRESS, CAMBRIDGE, MA, 1997.*

[13] *J.R. SEARLE: SPEECH ACTS. CAMBRIDGE UNIVERSITY PRESS, 1969, CAMBRIDGE MA.*

[14] *Y. LABROU. SEMANTICS FOR AN AGENT COMMUNICATION LANGUAGE. PHD THESIS DISSERTATION SUBMISSION, UNIVERSITY OF MARYLAND GRADUATE SCHOOL, BALTIMORE, SEPTEMBER, 1996.*

[15] *M.D. SADEK, P. BRETIER, V. CADORET, A. COZANNET, P. DUPONT, A. FERRIEUX, F. PANAGET. A CO-OPERATIVE SPOKEN DIALOGUE SYSTEM BASED ON A RATIONAL AGENT MODEL: A FIRST IMPLEMENTATION ON THE AGS APPLICATION. PROCEEDINGS OF THE ESCA/ETR WORKSHOP ON SPOKEN DIALOGUE SYSTEMS: THEORIES AND APPLICATIONS, VIGSO, DENMARK, 1995.*

[16] *ABROSE CONSORTIUM, AGENT BASED BROKERAGE SERVICES IN ELECTRONIC COMERCE, HTTP://B5WWW.BERKOM.DE/ABROSE*

[17] *JARON COLLIS, DIVINE NDUMU. ZEUS TECHNICAL MANUAL. INTELLIGENT SYSTEMS RESEARCH GROUP, BT LABS, SEPTEMBER 1999*

[18] *H. NWANA, D. NDUMU. A PERSPECTIVE ON SOFTWARE AGENTS RESEARCH. THE KNOWLEDGE ENGINEERING REVIEW, VOL. 14, ISSUE 2. CAMBRIDGE UNIVERSITY PRESS, SEP 1999.*

[19] *G. WAGNER, TOWARDS AGENT-ORIENTED INFORMATION SYSTEMS. TECHNICAL REPORT, FREIE UNIV. BERLIN, 1999. AVAILABLE AT HTTP://WWW.INF.FU-BERLIN.DE/~WAGNERG/INDEX.HTML*

[20] N. R. JENNINGS, K. SYCARA, M. WOOLDRIDGE, A ROADMAP OF AGENT RESEARCH AND DEVELOPMENT, AUTONOMOUS AGENTS AND MULTI-AGENT SYSTEMS, 1(1), 7-38, 1998.

[21] M. J. HUBER, JAM AGENTS IN A NUTSHELL. AVAILABLE AT HTTP://MEMBERS.HOME.NET:80/MARCUSH/IRS/

[22] THE HOMEPAGE OF FIPA-OS: HTTP://NORTELNETWORKS.COM/FIPA-OS/

[23] L. P. KAELBLING. A SITUATED AUTOMATA APPROACH TO THE DESIGN OF EMBEDDED AGENTS. SIGART BULLETIN, 2(4): 85-88.

[24] J. VEIJALAINEN, F. ELIASSEN AND B. HOLTKAMP, THE S-TRANSACTION MODEL, IN BOOK: DATABASE TRANSACTION MODELS FOR ADVANCED APPLICATIONS, MORGAN KAUFMANN PUBLISHERS, INC.", A. K. ELMAGARMID (ED.), 1992, PP. 467-513.

[25] A. LEHTOLA, T. LILLQVIST, K. KEKKI AND S. HAKKARAINEN, BUSINESS-TO-BUSINESS ELECTRONIC TRADING AS CO-OPERATIVE WORKFLOWS. IN BOOK: TECHNOLOGIES FOR THE INFORMATION SOCIETY: DEVELOPMENTS AND OPPORTUNITIES, J.-Y. ROGER & AL. (EDS.), IOS PRESS, AMSTERDAM, 1998, PP. 373-380.

[26] A. K. ELMAGARMID (ED.), DATABASE TRANSACTION MODELS FOR ADVANCED APPLICATIONS. ISBN 1558602143, MORGAN KAUFMANN PUBLISHERS, SAN MATEO, CA, 1992, 611 P.

[27] ROLF A. DE BY, WOLFGANG KLAS AND JARI VEIJALAINEN (EDS.), TRANSACTION MANAGEMENT SUPPORT FOR COOPERATIVE APPLICATIONS. ISBN 0792381009. KLUWER ACADEMIC PUBLISHERS, BOSTON, 1997, 222 P.

[28] A. SLADEK AND A. WOLSKI, MODELING INTER-ORGANISATIONAL WORKFLOWS. PROCEEDINS OF INTERNATIONAL SYMPOSIUM ON APPLIED CORPORATE COMPUTING (ISACC'96), MONTERREY, MEXICO, 1996, PP. 13-22.

[29] THE JKQML MANUAL. AVAILABLE FROM THE JKQML HOMEPAGE: HTTP://WWW.ALPHAWORKS.IBM.COM/FORMULA/JKQML

[30] H. JEON, C. PETRI, M.R. CUTKOSKY. JATLITE: A JAVA AGENT INFRASTRUCTURE WITH MESSAGE ROUTING. IEEE INTERNET COMPUTING, VOL. 4, NO. 2, MARCH/APRIL 2000.

[31] IKV++ GMBH, GRASSHOPPER PROGRAMMER'S GUIDE. AVAILABLE FROM HTTP://WWW.GRASSHOPPER.DE.

[32] IKV++ GMBH, GRASSHOPPER USER'S GUIDE. AVAILABLE FROM HTTP://WWW.GRASSHOPPER.DE.

[33] CRYSTALIZ INC., GENERAL MAGIC INC., GMD FOKUS, IBM, TOG. OMG JOINT SUBMISSION "MOBILE AGENT SYSTEM INTEROPERABILITY FACILITY", NOVEMBER 1997, AVAILABLE FROM FTP://FTP.OMG.ORG/PUB/DOCS/ORBOS/97-10-05.

[34] OBJECT MANAGEMENT GROUP. COMMON FACILITIES RFP3. REQUEST FOR PROPOSAL OMG TC DOCUMENT 95-11-3, OBJECT MANAGEMENT GROUP, FRAMINGHAM, MA, NOVEMBER 1995.

[35] THE GRASSHOPPER HOME PAGE: HTTP://WWW.GRASSHOPPER.DE.

[36] FOUNDATION FOR INTELLIGENT PHISICAL AGENTS, AGENT MANAGEMENT. FIPA 98 SPECIFICATION, PART 1. AVAILABLE AT HTTP://WWW.FIPA.ORG/SPEC/FIPA8A23.DOC

[37] OBJECT MANAGEMENT GROUP, AGENT TECHNOLOGY, GREEN PAPER. OMG DOCUMENT EC/00-03-01, MARCH 2000.

[38] ITU-T REC. X.701, INFORMATION TECHNOLOGY – OPEN SYSTEM INTERCONNECTION – SYSTEMS MANAGEMENT OVERVIEW, 1992.

[39] ABS CONSORTIUM, ARCHITECTURE FOR A BROKERAGE INFORMATION SERVICE, HTTP://B5WWW.BERKOM.DE/ABS

[40] E.ATHANASSIOU, I.TOTHEZAN, P.ALZON AND G.T.KARETSOS. "ENTERPRISE MODELLING OF INFORMATION BROKERAGE AND RETAILER SERVICES". FIRST INTERNATIONAL ENTERPRISE DISTRIBUTED OBJECT COMPUTING WORKSHOP (EDOC'97). MARRIOTT RESORT, GOLD COAST, AUSTRALIA. OCTOBER 1997.

[41] H.J.EINSIEDLER, P.BARRETT, D.CHIRICHESCU, M.-P.GLEIZES, P.GLIZE, C.HARBILAS, A.L?GER, J.I.MORENO, T.J.WILKE, "ABROSE : A CO-OPERATIVE MULTI-AGENT BASED FRAMEWORK FOR ELECTRONIC MARKETPLACE", INFOWIN-INFOBRIDGE: BOOK ABOUT "AGENT TECHNOLOGY", ACTS RELATED PUBLICATION, NOVEMBER 1999

[42] TAKADA YUJI, ICIKI HIROKI, OKADA MAKOTO, MOHRI TAKAO – A PROPOSAL ON AGENT BROKERAGE. FIPA CFP5-008, JANUARY 1999.