

Architectural Properties of Multi-Agent Systems

Onn Shehory

CMU-RI-TR-98-28

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

December 1998

©1998 Carnegie Mellon University

This research has been supported in part by the Army Research Lab under contract No. DAAL0197K0135 and the ONR grant No. N00014-96-1-1222.

Abstract

One aspect of multi-agent systems (MAS) that has been only partially studied is their role in software engineering, and in particular their merit as a software architecture style. As we demonstrate, multi-agent systems developed to date have several common architectural characteristics, even though differences in their design and implementation result in variations in their strengths and weaknesses. A large portion of the research in the design and implementation of MAS addresses questions such as: given a computational problem—can one build a MAS to solve it? What should be the properties of this MAS given the problem? Having developed a MAS, what is the class of problems that this MAS, either as developed or with slight modifications, can solve? MAS research has provided several answers to the questions above. However, more fundamental questions were left un-answered: *given a computational problem—is a MAS an appropriate solution? If it is, what type of MAS should be preferred?* Asking these questions (and answering them) should precede the previous ones, lest multi-agent systems may be designed and implemented where much simpler, more efficient solutions apply. In this report we provide analysis guidelines which—although do not explicitly answer these questions—support designers in their assessment of the suitability of MAS as a solution to computational problem they address. We discuss the architectural properties that should be considered when analyzing such systems and support our work with case-studies of several MAS.

1 Introduction

In the past few years Multi-Agent Systems (MAS) have emerged, combining research from the field of Distributed Artificial Intelligence (DAI) with a new approach to software engineering. This new paradigm proposes solutions to highly distributed problems in dynamic, open computational domains. The DAI research discusses issues of intelligence which include, among others, learning [11], negotiation [15, 24], strategic behavior [28, 25], social behaviors and norms [31, 3], cognitive activities (reasoning [10], beliefs, desires, intentions [23], emotions [1]) and suggests a variety of architectures to support such intelligent behaviors. These issues have been a major interest of MAS research, but are not the only ones. One aspect that has been only partially addressed is the role of MAS in software engineering¹, and in particular its merit as a software architecture style [27]. We find the latter issue to be of interest to both the AI and the software engineering communities, and focus on it in this report. As we demonstrate in this report, multi-agent systems developed to date have several common architectural characteristics, even though differences in their design and implementation result in variations in their strengths and weaknesses.

A large portion of the research in the design and implementation of MAS addresses the following questions:

- Given a computational problem - can one build a MAS to solve it? What should be the properties of this MAS given the problem?
- Having developed a MAS, what is the class of problems that this MAS, either as developed or with slight modifications, can solve?
- Can one devise a generic specification language or a protocol for MAS design and specification? If this is possible, can one develop a tool for translation of the specification into code?

Researchers in the MAS community have provided several good answers to the questions above (some of that work appears in [12, 13, 22]). However, more fundamental questions were left un-answered:

- Given a computational problem – *is a MAS an appropriate solution to it?*
- *If it is, what type of MAS should be preferred?*

Asking these questions (and answering them) should precede the previous ones, lest multi-agent systems may be designed and implemented where much simpler, more efficient solutions apply. Yet, due to their unique properties and the advantages that stem from these, we would like system designers to consider MAS as one of the possible solutions to computational problems in hand. To support designers in their assessment of the suitability of MAS to their problem, we discuss the architectural properties of MAS and study several specific MAS to demonstrate these properties, analyzing their advantages and drawbacks.

This report is organized as follows. First, in this section we provide a short introduction to MAS from a software architecture viewpoint (section 1.1), a brief introduction to software architecture

¹Software engineering issues such as specification and verification of MAS were lately discussed by Wooldridge [35].

(section 1.2) and some terminology that we use to refer to MAS (section 1.3). Then, in section 2, we present properties of multi-agent systems relevant to software architecture and design. We also assess the advantages and disadvantages of different styles of MAS. We proceed in section 3, where we study several multi-agent infrastructures and analyze them in light of the properties discussed in section 2. Finally, in section 4, we conclude and point at open questions and future directions of this research.

1.1 Software Architectural Facets of Multi-Agent Systems

We attempt to examine MAS mainly with respect to their software architecture attributes such as robustness, flexibility and adaptability, code re-usability, throughput etc. An analysis from this viewpoint is not common in MAS publications. We find it necessary to analyze the relation between the software architecture of a MAS and its functionality. This analysis should provide a system designer/engineer with information upon which she may decide both whether a MAS may be an appropriate computational solution to a problem in hand, and if so, what type of MAS provides the most appropriate solution for this problem.

Viewing them as a software architecture style, MAS are systems comprised from components, called agents. The agents are usually designed to be autonomous, where autonomy refers to a component not depending on the properties or the states of other components for its functionality. Nevertheless, to construct a meaningful multi-component system, these components are able to interact, usually by passing messages in a pre-defined protocol (agent communication language, e.g., KQML [6]). In contrast to distributed object architectures (CORBA [19]), it is commonly assumed that no direct function call or implicit event invocation between components (that is, the agents) are allowed. In particular, the autonomy of an agent a means that although others can request for a service s which is provided by a (in object-based systems, a request for a service s from a service provider a would be performed by calling a method of a 's), a has the sole control over the activation of its service s and may refuse to provide it, or ask for a (monetary) compensation for its service. (In object-oriented systems, a public method of an object x can be activated by any other object y , and x is not assumed to have any control over this activation.)

When attempting to provide a solution to a computational problem, the designer, usually upon the analysis of the problem and its characteristics, tries to recognize typical patterns. These are compared to her knowledge of similar patterns of previous problems and for common software architectures. Applying the most appropriate one will reduce the time for development and increase the efficiency and adequacy of the provided computational solution. While several architectural styles have been recognized and described in the literature, multi-agent systems have not yet gone through this process. Agents are already proliferating in cyberspace and multi-agent systems are moving from the laboratory to the field. Due to their unique suitability to several classes of computational problems, it is important to characterize MAS as a software architecture style and to provide the architectural specifications of such systems. Such specifications will equip system designers with a family of appropriate solutions for highly distributed problems in open, heterogeneous, dynamic and information-rich environments. In this report we make a first step towards this goal.

1.2 Introduction to Software Architecture

Software architecture (SA) is an important discipline within software engineering. As a result of the increase in size and complexity of software systems, specification and design of the overall structure of such systems becomes an increasingly dominant part of their development. At the abstract level, SA involves the description of components from which systems are comprised, the interaction among these components and the patterns according to which the components are combined to form the whole system. At the practical level, SA refers to the design and specification of issues such as component decomposition and organization, communication protocols, control and data flow and structure, synchronization and access to data etc.

To differentiate between architectural styles, software architecture usually employs some common framework. The framework we adopt in this study is based on treating a system as a collection of components and a set of interactions between these components. The framework determines what the components that construct instances of the architecture style and what the constraints on the ways in which these components can be combined are. These may include, among others, constraints on the topology of the system (for instance - can an agent be subsumed by another agent), or constraints on the semantics of execution. Once the framework is used to describe styles and systems, one can have a better understanding of the underlying computational model. This can be used to sort out the essentials of the style. It also supports comparison between styles and between systems within the style, thus evaluating advantages of disadvantages of the style.

To utilize this approach, it is essential for our cause to distinguish unique components, interactions and constraints of different MAS, thus providing a tool for the assessment of advantages and drawbacks of different styles and sub-styles. For this we first need to define a terminology, so that the reader will have a coherent view of terms to which we later refer.

1.3 MAS Terminology

Being a young field of research, multi-agent systems suffer from the lack of an agreed upon terminology for describing systems, components and the relationships between them. We define several terms to be used throughout this report.

- *Agent architecture* - describes the modules from which a single agent is comprised, the relationships between, and the interactions among these modules. For example, agents (in the context of MAS) usually have a communication module to enhance communication with users and other agents. Some types of agents also have a planning module. Commonly, incoming messages arriving at the communication module will affect the planning module by some connection (and with some restrictions), and the planning module may create outgoing messages to be handled by the communication module.
- *Multi-agent organization* - describes the way in which multiple agents are organized to form a MAS. Relationships and interactions among the agents and specific roles of agents within the organization are the focus of multi-agent organization. The agent architecture is not part of the multi-agent organization (although inter-relationships between the two are common). For instance, the agents may be organized in a fixed hierarchy, where the inter-relationships are pre-defined, thus reducing the need to locate others and reason about them, and the amount of communication necessary for the functioning of the system.

- *Multi-agent infrastructure* - describes the agent architecture and the multi-agent organization, and possibly the dependencies between the two (when present), thus provides an infrastructure that enables constructing domain-specific MAS. It is also common practice to include in the infrastructure services which are provided to enhance MAS activity and organization such as agent naming service, agent location service (e.g. directory service), etc. The infrastructure may include services, either necessary or optional, for the activity of the MAS.
- *Multi-agent infrastructure services* - include services that are provided with the MAS infrastructure to support a variety of system needs. The following services can be found (or are desired) in MAS:
 - System design and development tools (e.g., agent editor, syntax checkers, system correctness verification tools).
 - System (dynamic) organizational activity enhancement, e.g., agent location and coordination mechanisms (may be supported by e.g., middle agents).
 - Tools for increasing system efficiency in resource utilization (e.g., network sensing, mobility enhancement).
 - Agent and MAS activation, interfacing and testing tools.
 - Securing transactions of information and electronic goods/money (via, e.g., security protocols and certification authorities).

The terms above will be used and elaborated upon in the following sections.

2 Architectural Attributes of MAS

In this section, we present and evaluate the architectural properties of MAS. This presentation should allow for comparison between, and assessment of, different MAS infrastructures. Based on the attributes discussed here, we present, in proceeding sections, case studies where we describe and analyze several multi-agent systems.

2.1 Agent Internal Architecture

In the last years, a large variety of agent internal architectures were introduced by agent researchers and developers. Some of these architectures are presented later in this document. Yet, when referring to agents which are part of, or can be incorporated into, a multi-agent system, this number decreases significantly. This is because, for agents to be incorporated into MAS, it is necessary to equip them with components that will allow for meaningful interaction with other agents and users (e.g., a communication component) and some restrictions on the way in which these interactions are performed. Regardless of their internal architecture, agents in the context of MAS should be able to perform tasks or provide services. Ideally, one would prefer all the details of an agent's internal architecture to be hidden from other agents and users. This would allow entities with which the agent interact to assume some capability of the agent and some interaction protocols (and maybe additional qualitative assumptions), but will prevent the need that they know what methods and

components are employed by the agent to perform its tasks and provide its services. To date, this aim is not achieved yet. Agents within MAS usually do assume some type and structure with respect to the agents with which they interact. Nonetheless, at the time of writing this report MAS inter-operability research is performed in several labs to overcome this limitation. Another aspect of internal agent architecture is its influence on the overall MAS behavior and the ability of the MAS to efficiently perform its tasks. Although such influence seems an inevitable property of MAS, no extensive comparative research was performed to investigate this issue.

2.2 Multi-Agent Organization

Broadly speaking, MAS are organized in one of the following ways: hierarchy, flat organization (sometimes referred to as democracy), subsumption, and a modular organization. Hybrids of these and dynamic changes from one organization style to another are also possible, though not very common in implemented MAS (probably due to the complexity of implementing dynamic re-organization). We summarize below the properties of these MAS organizational models.

- Hierarchical MAS (e.g., federated MAS) are organized such that agents can only communicate subject to the hierarchical structure. The advantages of this restriction is that there is no need for a mechanism for agent location. For example, in section 3.6, where we present a federated MAS, the components in the upper level of the hierarchy, the facilitators, are in charge of agent location. Another advantage of hierarchical structure in MAS is the significant reduction in the amount of communication in the system. The disadvantages of this organization is the strict structure, which does not allow agents to dynamically organize themselves to best fit the needs of a specific task. Also, usually the hierarchy implies that the lower levels depend on the higher levels (as in, e.g., OAA [17]), and higher levels may even be in partial or full control of the lower levels. This may be in contrast to a requirement for autonomy of agents or for their being self-interested. A hierarchical organization may also imply, to some extent, a centralized control, which is undesirable in systems which are comprised from components that belong to different organizations, and may be geographically distributed as well. An example of a MAS with a hierarchical organization is presented in section 3.6.
- A flat organization of a MAS implies that each agent can directly contact any of the other agents. No fixed structure is applied on the system, however agents may dynamically form structures to perform a specific task. In addition, no control of one agent by another agent is assumed. Such an organization requires that either the system is closed, so that each agent knows about all of the others ahead of time, or (when the system is open, as is e.g. RETSINA [32]) an agent location mechanism must be provided as part of the infrastructure. A flat organization is advantageous since it fully supports autonomy and self-interest of agents as well as distribution and openness of the MAS. It also allows for dynamic adjustments of the MAS organization to changes in tasks and environment. These openness and dynamism, however, result in communication overheads, the need for agent location mechanisms as well as mechanisms for dynamic MAS re-organization. The amount of reasoning an agent performs with regards to other agents (and consequently the local computational overhead of

an agent) increases significantly in a flat organization. An example of a flat MAS organization is presented in section 3.7.

- There are MAS where some agents are components of other agents. These agents are subsumed by the container agents, which in turn may be components of larger container agents. The subsumption model is somewhat similar to the hierarchical model, however it takes it to the extreme by requiring that the subsumed agents completely surrender to the control of the container agent. From a software architectural viewpoint, such architecture resembles a inclusion of objects within a larger object, except for the (important) difference in the control methods. That is, while objects are usually controlled and activated by (possibly remote) procedure call or by event invocation, agents are activated by high-level communication, i.e., message transmission. The strict control relationships in the subsumption organization results in efficient tasks execution and low communication overhead, however restricts the system to address a well defined set of tasks, with virtually no flexibility and adaptability. It is also not simple to modify a subsumption MAS (e.g., add a new component) in the face of long-term changes in tasks and environment of the system. An example of a MAS with a subsumption organization is presented in section 3.3.
- A MAS has a modular organization when it is comprised from several modules, where each of these modules can be perceived as a virtually stand-alone MAS. Typically, the partition of the system into modules is done along dimensions such as geographical vicinity or a need for intense interaction among agents and services within the same module. Often, the system is comprised of such parts as a result of its development process, during which new modules were gradually added to an already existing system. Modularity increases efficiency of task execution and reduces communication overhead. Also, within each module high flexibility, similar to flat organization flexibility, is usually enabled. On the other hand, re-organization across modules is rather complex, thus flexibility is limited. In addition, the given modularity implies constraints on inter-module communication. For instance (see OSACA, section 3.2), while intra-module communication is usually connection-oriented, inter-module communication may be connectionless, which prohibits the execution of tasks that require inter-modular concurrency.

In addition to organizational issues and combined with them, other MAS properties play a role in their architecture and affect their performance. Among them one can find communication structures and protocols, degree of system openness, level of flexibility, infrastructure services and system robustness. These are summarized and discussed below.

- **Communication:**

A large number of MAS use a specially designed communication protocol, that best fits their agent architecture, MAS organization and the typical tasks of these systems (e.g., ARCHON, section 3.1, or OAA [17] which uses an agent communication language developed specifically for OAA agents²). The advantage of such protocols is in their efficiency: the agents transmit only the information necessary with very little overhead and message packaging and parsing. On the other hand, such systems lack the ability to converse with agents which are not

²An OAA agent that can “speak” KQML is in a development stage at the time of writing this report.

implemented using the same communication infrastructure, and it is most unlikely that other agents will have these specialized communication protocols implemented in them. To overcome such limitation, there is a trend to provide agents with communication protocols which are more generic. Several MAS support agent communication languages (ACLs) such as KQML and FIPA³. This, however, does not mean that two agents from different systems that support the same ACL, e.g., KQML, are able to understand one another. Inter-operability requires a common ontology as well. To date, this issue is not yet resolved. Nevertheless, communication modules that are generic in nature are developed (e.g., in RETSINA and D'Agents [8]).

Distributed computational systems implement several standard communication protocols. We distinguish three main attributes of such protocols which are relevant to MAS.

1. Symmetry:

In several MAS, client/server protocols are used for communication. Since client/server protocols are well supported and documented as part of operating systems and programming languages, such implementations are simple and efficient. The drawback of client/server protocols is that they imply asymmetry between the communicating entities: one is in control of the communication whereas the other party can only respond upon request, and cannot initiate communication. Designers of MAS, especially open MAS with a flat organization, have realized that such asymmetry is inappropriate for these systems, and implemented symmetric means of communication. This, however, increases protocol complexity and slows down communication.

2. Message recipients:

Messages in a network may be sent to a single addressee, to multiple ones (multicast) and to all (broadcast). In an open system, broadcast is impractical, since an agent does not know all of the other agents. Therefore, open MAS usually implement peer-to-peer or multi-cast communication. In closed MAS, however, broadcast is commonly used (e.g., in ARCHON, section 3.1). The advantage of it is in the simplicity of the protocol. The disadvantage is that all of the agents receive the message, even when it is completely irrelevant for them, thus increasing network congestion.

3. Connection type:

Connection-oriented and connectionless communication are both implemented in MAS. The advantages and drawback of these are not unique to MAS, and can be found in standard networks' text books (e.g., [33]). Typically, MAS implement connection-oriented communication, however in some cases connectionless protocols are supported as well. Connection-oriented communication is preferred when dependent tasks are performed concurrently by multiple agents, and coordination is necessary during execution. In such situations connectionless communication may prohibit coordination and proper task performance. In MAS where task execution is loosely coordinated and where concurrency is of minor importance, connectionless communication is sufficient.

³FIPA stands for Foundation for Intelligent Physical Agents. FIPA provides a specification for agent-based applications including, among other specifications, an ACL nicknamed FIPA as well.

- **System openness:**

The openness of a MAS refers to the ability of introducing additional agents into the system in excess to the agents that comprise it initially. While some MAS do not allow the addition of agents (at all), others may be more open, allowing to add agents with different styles of addition. In its basic level, MAS openness refers to the OSI definition of system openness. However, in MAS, additional properties are considered. One can sort openness of systems into three broad categories:

1. **Dynamic openness:** In MAS, the level of dynamism allowed for adding new agents has a significant effect on the properties of the system. MAS that allow agents to leave or enter the system dynamically, during run time, without any explicit message to all of the other agents in the system, are the most open ones. The advantage of such openness is in the ability of the system to dynamically adjust itself to changes in the environment, tasks, and availability of capabilities and resources. This type of dynamism is important for MAS that are deployed in environments with high levels of uncertainty. A major disadvantage of this extreme openness is the required additional services and computation. When agents can unpredictably appear and disappear, a robust agent location mechanism is a must. Also, agents must be provided with methods to alternate their tasks execution and planning, since availability of necessary capabilities and resources varies.
2. **Static openness:** Less dynamic, yet considered open, is the case where agents can be added to the system without re-starting it, but either all of the agents are notified on such an addition, or they all hold in advance as list of prospective additional agents. This type of openness eliminates the need for an agent location mechanism, and reduces the complexity of contingent execution and planning computation (although these are not eliminated). On the other hand, the flexibility of the system and its ability to adjust itself to dynamic changes is restricted. Such openness is insufficient for environment with high levels of uncertainty. It would better fit cases where changes are more gradual and predictable.
3. **Off-line openness:** The most restricted type of openness is the one that allows the addition of new agents only off-line, by halting the system, adding agents, updating some connection information, and re-starting the system. This approach allows for changes in the system over time, however no dynamic ones are supported. On the other hand, there is no need for infrastructure services nor for additional computation to handle dynamic changes in the system. Hence, such systems will perform more efficiently in cases of well defined, predictable problem domains.

The categories of MAS openness presented above can be shifted towards one another, thus gaining some advantages and compromising on others.

- **Infrastructure services:**

Infrastructure services are, in some MAS, inseparable from the system, whereas in others they are optional or completely unnecessary. We provide details of some of these services:

1. An open MAS must be provided with an agent naming service, so that no two agents will have identical names, and the consequent confusion be avoided. Close systems or slightly open systems, where all of the agents (or the possible ones—in the latter systems) are known in advance do not need a naming service.
2. Another type of service necessary in an open MAS is an agent location service (e.g., brokering or matchmaking). When the existence and availability of agents are not common knowledge, this service is a pre-condition to the ability of a MAS to perform its tasks. An agent location service is sometimes implemented in a centralized manner, which is simpler to implement and maintain, however more vulnerable and creates a single point of failure of the MAS. Conversely, distributed location mechanisms (see, e.g., [14]) are much more complicated to design, implement and maintain, and increase communication and computation overheads, however provide a reliable, robust service.
3. An optional service which is mostly helpful in open MAS is a security service. In an open MAS, an agent may be uncertain with regards to the true identity and the trustworthiness of other agents. Security mechanisms can reduce the risks that stem from this uncertainty. To date, only few MAS provide security services as part of their infrastructure. The addition of such services may require introducing trusted third parties such as electronic Certification Authorities (CAs) as well as implementing protocols to be followed by the agents. This, inevitably, increases computation (e.g., for encryption and decryption) and communication overheads, and may create bottlenecks at the third parties.
4. There is a unique family of MAS—those that allow for agent mobility (e.g. Agent Tcl [9]), where an infrastructure service that supports mobility may be required. The most common way to provide this service is via mobility servers, sometimes called agent docks. Agent docks are servers which are running on machines where mobile agents are allowed to arrive. The mobile agent “docks” at the dock, and the dock provides interface and access to resources on that machine subject to the restrictions applicable to the arriving agent. Mobility servers increase computation overheads on the machines they run. On the other hand, they provide an essential service in case that mobility is necessary.

Other system services may be present as well (e.g., a shared database), but are not as common or necessary as the ones presented above.

- **System robustness:**

One of the advantages of MAS is the distribution of execution, which allows for increase in overall performance. In addition, failure of one agent does not necessarily imply a failure of the whole system. The robustness provided by MAS is further increased by replicated capabilities. This replication is enabled by having multiple agents with same or similar capabilities. In such cases, when an agent that has some capability becomes un-available, another agent with a similar capability may be approached. Replicated capabilities are more natural (and useful) in open MAS, however can support robustness in close MAS as well. The disadvantage of this replication is in the resulting redundancy. The robustness of a MAS

depends also on the type of services it uses and the way in which these are implemented, as mentioned in items above.

Although the properties discussed above are not unique to MAS, combining them in a single system is unique to MAS. This combination results in the suitability of MAS for solving problems where information, location and control are highly distributed, heterogeneous autonomous (self-controlled) components comprise the system, the environment is open and dynamically changing, and uncertainty is present. Notice that if only a few of these problem domain characteristics are present, it may be advisable to consider other architectures as solutions, instead of MAS. One should bear in mind that the limited development tools for MAS, the high complexity of such systems and the amount of code replication in them may result in excessive, unnecessary efforts in the development phase as well as inefficient solution and poor system performance.

Other software-architectural properties, although important, are of lesser significance for the design of multi-agent systems.

3 Case Studies of MAS Infrastructure

In this section we present several MAS, concentrating on the multi-agent infrastructure attributes of these systems. We analyze their properties along several dimensions, emphasizing the agent architecture and the multi-agent organization. We discuss the advantages and draw-backs of different MAS infrastructures in light of the generic MAS properties presented in section 2. We also explain what the benefits of using such a software architecture style are and characterize the types of problems to which such an architecture can be applied and provide solutions, as well as the advantages and disadvantages of such application. We emphasize the unique architectural properties of MAS and the advantages that stem from these properties.

3.1 The ARCHON Infrastructure

Archon is an infrastructure for multi-agent systems [34]. It provides a system organization as well as agent internal architecture. Archon operates in the domain of industrial process control, and is aimed at reducing the complexity of control in large, complex (usually pre-existing) computational systems embedded in such domains. This is achieved via the distribution of execution and control. The underlying assumption is that there are existing domain-specific sub-systems which provide the solutions to primary application problems. The functional requirements of the whole system, which is comprised from these, are determined in terms of these component systems. In order to achieve cooperation among the component systems, Archon provides a layered organization, somewhat similar to the OSI layered communication protocol. A session layer supports inter-connection services between agents. An Archon layer is attached to each component system (see figure 1). The Archon layer component provides two interfaces: one to its underlying application system (by its monitor module) and one to the rest of the agent community (by its communication module) via the session layer. Each Archon layer component controls itself, its application system, and its interaction with other agents. In the Archon approach an agent is the combined entity that includes the application system component and its attached Archon layer. The application systems are considered intelligent systems (IS), and the Archon layer does not add to it any domain specific

functionality or intelligence. It serves only for coordination and cooperation among domain-specific components.

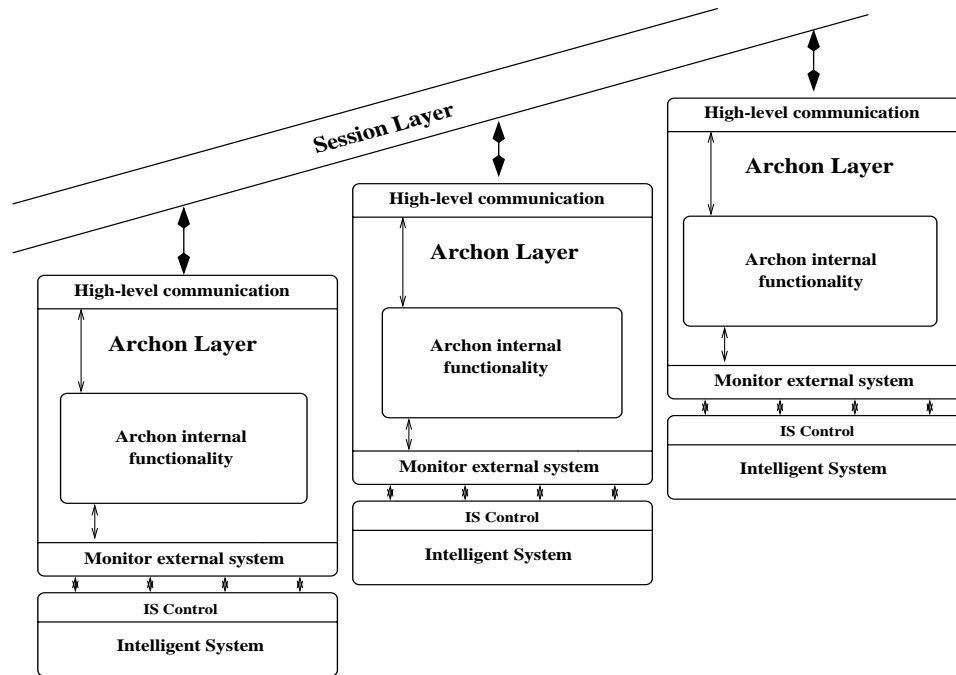


Figure 1: The Archon multi-agent organization.

The designers of the Archon architecture specified it to be an open architecture. That is, more applications may be added to the whole system, by simply attaching an Archon layer component to the added domain-specific component. The Archon view of openness complies with the OSI approach to openness, however, it is somewhat limited. Firstly, the addition (or disconnection) of components cannot be done dynamically. From the description of the system in [34] it follows that the ability of agents to locate (and communicate with) others is provided by two complementing methods: (1) Hard-wired addresses are given in a local address-list of each agent; (2) Agents broadcast their availability to a single matching services agent, well-known to all agents, which serves as a blackboard-like mechanism. The latter method, (2), is a centralized mechanism which results in a single point of failure (which solutions involving multiple agents attempt to avoid). The first method (no. 1) limits the openness of the system since only well known agents can be added to it, or at least all of the addresses of agents that may potentially join the system must be known in advance. This means that new agents that appear dynamically cannot add themselves or be added to the system (unless they are somehow known to the system members). In the case of dynamic multi-agent systems, such an organization is somewhat closed. Secondly, even when agents are known in advance, their ability to join the system depends on their ability to appear as Archon agents. That is, an agent can be connected to an Archon-based MAS only by using the Archon session layer, which requires following its communication protocols. Note that this confining attribute is typical to many multi-agent infrastructures, mainly because there is no standard protocol for agent communication.

An Archon agent architecture is comprised from two components: the Archon layer and the

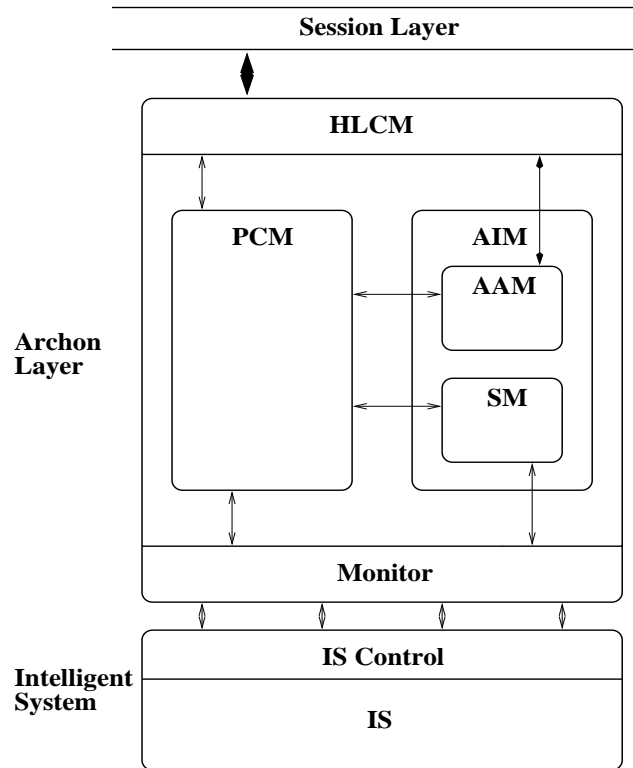


Figure 2: The Archon agent internal architecture.

domain-specific, Intelligent System (IS) which the Archon layer monitors. The IS is usually a previously-existing, separately designed, developed and implemented component, which does not follow a dictated Archon architecture. The internal architecture of an Archon Layer consists of several modules, as follows (see figure 2).

- The *information management module* (AIM) provides a model and a language for information manipulation, for local and remote access and update. It subsumes two internal modules, the SM and AAM (both described in the following items).
- The *self model* module (SM) holds a model of the IS and performs reasoning about its state. The self model contains both state record and plans for controlling the IS. The plans are used by the monitor and can be updated by other modules.
- The *agent acquaintance module* (AAM) contains models of other agents in the community. The model contains static information (e.g., capabilities of other agents) and dynamic information (e.g., the current state of another agent).
- The *monitor* monitors the IS by checking the states of tasks that may be active in that system, receiving requests for information from the IS and forwarding them to the other internal modules of that Archon layer (and agent). It also starts and stops specific tasks and supplies to the IS data from other agents (or users).

- The *planning and coordination module* (PCM) performs situation assessment, planning and monitoring of cooperation with other agents. It initiates cooperation with others and responds to cooperation requests.
- Viewing Archon as a layered architecture, the *high level communication module* (HLCM) is defined as the layer between the session layer and the other Archon modules (collectively referred to as the Archon layer). The HLCM provides the other modules with three key services: intelligent addressing, filtering and message scheduling. The HLCM directs outgoing messages from the Monitor and PCM to other agents and filters incoming messages relying on the acquaintance models in the AAM.

To summarize, Archon is a multi-agent infrastructure with a flat organization and static openness. It allows for cooperation between previously-existing specialized systems. It supports distributed control of these systems as well as high level communication and information exchange among them. In addition, Archon enables adding new specialized sub-systems to the combined system without re-compiling the system. This addition is limited to previously-known names and addresses of the components to be added. Archon is designed as a layered architecture. Conceptually, each layer may be replaced by a component that complies with the interface requirements of the adjacent layers.

3.2 The OSACA and DIDE Infrastructure

OSACA, an Open System for Asynchronous Cognitive Agents [26], is a general multi-agent infrastructure. DIDE, a Distributed Intelligent Design Environment, takes advantage of this infrastructure [29] to address a specific type of problems, i.e., long-term complex product design. DIDE is aimed at allowing for the building of a truly open system, that is, a system to (from) which users can freely add (remove) agents without having to halt the system or re-initialize work in progress. The system should utilize asynchronous agents to aid users in information sharing, action coordination and search for globally near-optimal solutions for the design problems they encounter. According to DIDE, agents are assumed to be connected to existing engineering tools or data/knowledge base systems, or to user interfaces. In OSACA, all of the agents are connected to a network, and each agent can reach any other active agent by means of broadcasting messages. All agents receive messages which they may or may not understand. In the former case they perform the work related with the message, whereas in the latter they simply ignore the message. A new agent is introduced to the system by connecting it to the network. In DIDE, each agent offers a specific service, usually by encapsulating an engineering tool. Interaction among agents is based on a shared communication protocol and a language for information and service request. For local network communication DIDE utilizes ToolTalkTM, and for Internet communication structured e-mail messages are used. A basic underlying assumption in OSACA and DIDE is that broadcasting a message to all of the agents is possible. On local networks, ToolTalkTM supports this type of communication. However, for working in an open Internet, this may require that agents have the information about the existence and the addresses of all other agents.

Task execution in OSACA is based on local initiation. When an agent perceives the necessity of help from others (either services or information), it either broadcasts the need to all, or multi-casts

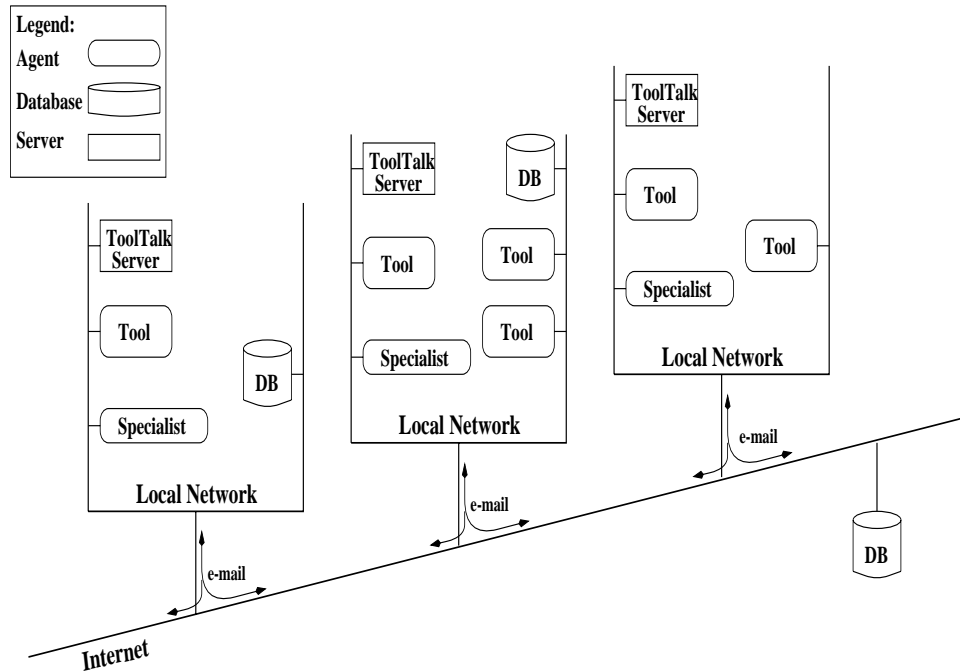


Figure 3: The OSACA-DIDE multi-agent organization.

to well-known specialists. In both cases it may either let all work on its request or contract it out to the most appropriate contractee.

The DIDE multi-agent organization is depicted in figure 3. In the figure tools refer to tools encapsulated in agents and specialists refer to agent who are either user interfaces for human specialists or agents who specialize in some administrative, organizational activity of the system. One can observe that the separation to local networks with better communication services (using the ToolTalkTM servers) may result in somewhat hierarchical organization (as a result of advantageous local communication). The OSACA and DIDE infrastructure, however, is rather open, since it does not prohibit the addition of new agents to any local network, and via these they may become known to the whole agent community.

3.2.1 The DIDE Internal Agent Architecture

An OSACA agent is autonomous in the sense that it can function independently from other agents. The agents each have their own deduction, storage and communication capabilities. The internal architecture is described in figure 4 (as appears in [29]). The agent is composed of following components:

- Symbolic models of other agents and methods for using them.
- A model of its own capabilities and domain-specific expertise.
- A module for handling tasks, modeling them dynamically and maintaining the information relevant to them.

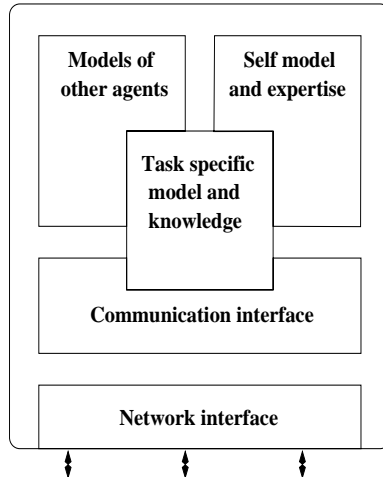


Figure 4: The OSACA-DIDE internal agent architecture.

- A communication module includes high-level communication, specific to the agent system. Handles I/O and exceptions as well.
- A network interface module to handle low-level communication.

In summary, OSACA and DIDE are a multi-agent infrastructure that, similar to ARCHON, allows for cooperation between previously-existing tool. It supports information and service exchange among the tools as well as user interfacing and system administrative services for users and tools. Openness (partly dynamic) enables to add new tools to the system without halting or re-starting work in progress. Since OSACA and DIDE use broadcast and multi-cast for communication, the addition of new tools is either limited to previously-known names and addresses of tools (in the multi-cast case), as in ARCHON, or result in a large communication overhead (in the case of broadcast). OSACA-DIDE provides a modular MAS organization. The organization of sections of the MAS in local networks benefits the OSACA-DIDE organization with an increased overall system openness. It also allows for better local network communications, however Internet communications are poorly supported. In addition, the separation to local networks and the superior local communication imply that cooperative activity is more likely to take place within local networks. This property limits the dynamism of cooperative task execution, although may increase efficiency in cases where such dynamism is not required.

3.3 The ADEPT Infrastructure

ADEPT (Advanced Decision Environment for Process Tasks) [21] is a multi-agent infrastructure that was developed for the management of business processes. A business process is the process that coordinates the collaboration of (semi-) autonomous units of a single or several organizations. This is done by means of task selection and decisions for service and product generation. In ADEPT, the MAS is comprised of agencies, where each agency may either be a single agent or, recursively, a collection of several agencies (i.e., a subsumption organization). Communication and cooperative task execution are performed either within an agency, among its members, or between

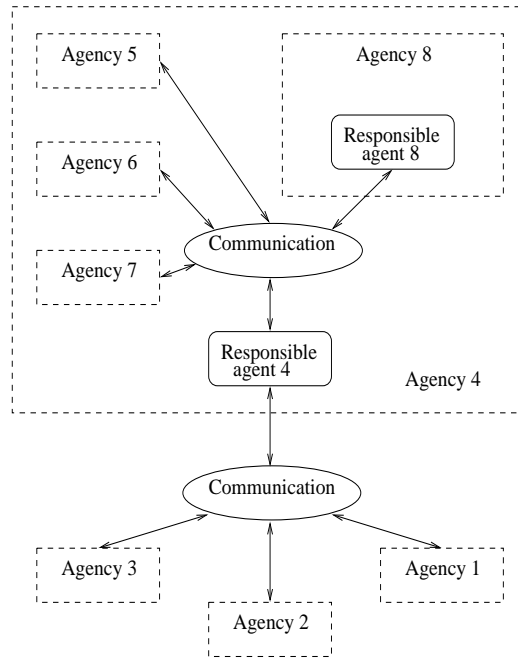


Figure 5: The ADEPT multi-agent organization.

agencies, however not directly between members of an agency and agents or agencies outside this agency. Each agency is represented by a single responsible agent. This means that the ADEPT architecture can support hierarchical and flat organizational structure as well as a combination of these, however a specific organization style must be decided upon in advance and cannot be modified dynamically. This organization is more flexible than the organization of the systems presented previously. However, the partition to agencies does not allow for dynamic changes in the structure of the organization and the grouping of agents on demand.

In the ADEPT system each agent provides a service. A service corresponds to either an atomic task or a composition of other services (which are provided by other agents). For each request for a service an agent decides whether to use the capabilities and resources of its own agency or to request services from other agents.

The ADEPT system organization is depicted in figure 5. Agents and agencies can only communicate and (directly) cooperate with agents and agencies within their encapsulating agency. For example, agencies 5-8, which are sub-agencies of agency 4, cannot communicate with agencies 1-3. They can only use the responsible agent 4 to contact entities external to agency 4 (however they are not assumed to know these entities). In ADEPT, communication requires that agents, agencies and tasks, which are all objects, register themselves with an Object Request Broker (ORB) as defined in the specifications of CORBA [19]. For this DAIS, a commercial implementation of the CORBA specification, is used. DAIS distributes messages between registered objects. The ORB receives requests from agents or tasks and sends a message to another agent or task. The broker is responsible for locating the object referred to by a requester and delivering the requester's message to this object. DAIS supports registration, de-registration and message delivery via its ORB.

3.3.1 ADEPT Internal Agent Architecture

The internal architecture of an ADEPT agent is broadly based on the ARCHON internal architecture. An ADEPT agent is comprised of several modules, as described below and depicted in figure 6.

- The *self model* module (SM) models the services the agent can provide, the resources available to it, its current schedule and other self knowledge. (This component is similar to the SM of ARCHON)
- The *acquaintance module* (AM) models capabilities of, and historical encounters with, other agents in the community. (This component is similar to the AAM of ARCHON)
- The *situation assessment module* (SAM) maintains the agent's schedule. Upon information concerning requests for services, commitments to the provision of services, failed tasks and other relevant information, it plans for negotiation and service execution and interacts with the SEM and IMM (see below) with respect to these plans. (This component is similar to the PCM of ARCHON).
- The *service execution module* (SEM) initiates and monitors services the agent has committed to provide and invokes services provided to the agent by others (based on the schedule and plans provided by the SAM). The SEM may re-execute failed services or notify the SAM for rescheduling or re-negotiating this service. The SEM communicates with other agents (using the CM).
- The *interaction management module* (IMM) performs negotiation for service requests and provision with other agents. It is prompted by the SAM and refers back to the SAM for scheduling information when representing the interests of the agent with respect to providing services to others. The IMM communicates with other agents using the CM, as does the SEM.
- The *communication module* (CM) provides the other modules with communications services that include message packaging, sending, receiving, interpreting and forwarding. The packaged messages must be transmittable through DAIS. The CM directs outgoing messages from the SEM and IMM to other agents (via DAIS) and filters incoming messages relying on the acquaintance models in the AM. Message failure is reported to the IMM.

The ADEPT multi-agent infrastructure utilizes a well-known agent internal architecture—the ARCHON agent architecture—with some modifications, however provides a more flexible organization than ARCHON does. It supports hierarchical, flat and combined organizations (whereas the organizations discussed previously support only one of these). Yet, dynamic changes of the organization are not supported. An interesting property of the ADEPT infrastructure is of tasks being autonomous entities. This contributes to the mobility of tasks among agents and agencies, thus may increase the efficiency of task re-distribution. Issues of openness are not explicitly discussed in ADEPT, however it does not seem to allow for more openness than the ARCHON infrastructure does.

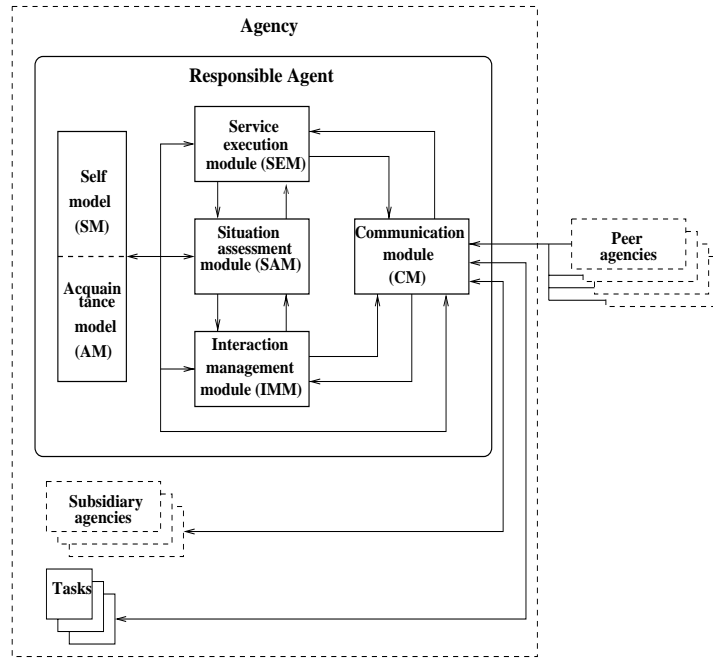


Figure 6: The ADEPT internal agent architecture.

3.4 The DESIRE Architecture

DESIRE is a framework for DEsign and Specification of Interacting REasoning components [2]. The DESIRE framework was used for developing reusable generic models for specific types of multi-agent applications. DESIRE provides a logical theory which is based on a temporal approach to the formal semantics of reasoning. Using this logic, DESIRE enables specification of generic multi-agent models as well as verification and validation of these models. The modelling framework of DESIRE consists of the following:

- A compositional design method for MAS.
- A formal specification language for conceptual as well as detailed system design, and software tools to support this design.
- An automatic specification-to-code translator.
- Tools for verification of static properties: consistency, correctness, completeness.

DESIRE specifications are based on compositional architecture. That is, an architecture composed of components with hierarchical relations between them. As common in object oriented design, each component has its input and output interfaces specifications defined and known to other components, whereas the internal structure is hidden from the rest of the system. This allows for component re-use, however the flexibility of this re-use is somewhat limited, since primitive components may be strongly tied to domain-specific knowledge and functionality.

DESIRE classifies agent types with regards to the way in which they interact with the environment and their internal reasoning methods. Agent types for which a DESIRE specification was

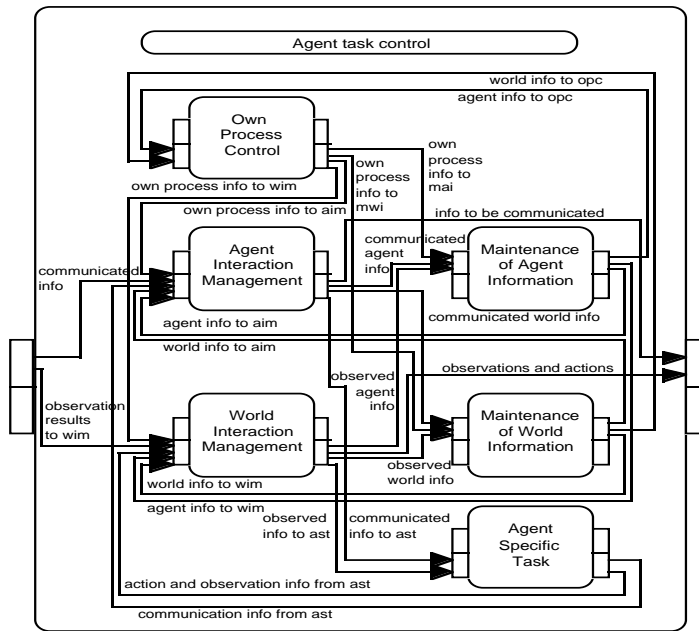


Figure 7: The DESIRE weak agent internal architecture.

used include reflective, reactive, cognitive, social and BDI (believe, desire, intend) agents. This classification, although important for DAI research, has a limited significance when software architecture properties are examined. While reactive and pro-active activity of agents can be referred to as software architecture issues, cognitive and social behaviors are not so. Nevertheless, the compositional approach of DESIRE presents interesting architectural properties. We present one agent type, referred to as *weak agent* [23], to demonstrate this architecture in figure⁴ 7.

A weak agent is autonomous, pro-active, reactive and social. These characteristics are supported by its internal components:

- Own Process Control (OPC) supports autonomy and pro-activeness.

⁴The figure was taken from a DESIRE publication.

- Maintenance of World Information (MWI) and World Interaction Management (WIM) support reactivity and pro-activeness with respect to the external world.
- Maintenance of Agent Information (MAI) and Agent Interaction Management (AIM) support social abilities as well as reactivity and pro-activeness with respect to other agents.
- Agent Specific Tasks (AST) performs the specific tasks of the agent.

Following its compositional approach, the DESIRE framework does not dictate a specific agent organization. It allows for a variety of organizations, where each is designed to best fit the properties of a specific problem domain. This results in flexibility in the design stage of MAS based on DESIRE, however once such a system is designed its organization is no longer flexible or dynamically adaptable. This is since the components of such a system cannot re-arrange themselves to address dynamic changes in the environment and the tasks to be performed. Yet, one should note that dynamic flexibility is commonly traded off with efficiency, and the DESIRE framework seems to prefer the latter.

To summarize, the DESIRE framework provides a generic infrastructure for developing MAS. This infrastructure consists of semantics and a logic for specification, verification and validation of MAS, their components and the interactions among these components. The compositional approach of DESIRE follows common trends in object-oriented design, taking it to a higher level of abstraction. By this, re-usability of code via the re-use of components is allowed (mainly for high-level components which are not strongly-tied to a specific domain). DESIRE provides a detailed internal agent architecture, and leaves the agent organization to be decided upon by MAS designers. Thus, flexible design is supported, however flexibility during MAS operation is not supported.

3.5 The MACRON Infrastructure

MACRON (Multi-agent Architecture for Cooperative Retrieval ONline) [5] is a multi-agent infrastructure that was designed for cooperative information gathering. In MACRON, user queries are translated to information gathering plans. These plans are executed by multiple agents. Information sources are assumed to be available on the WWW. The agents in MACRON form a matrix organization (as shown in figure 8), which is a dynamic, multi-hierarchy organization. The system consists of Query Manager (QM) agents, functional units (FU) with a functional manager (FM) for each unit, a farm of low-level information retrieval agents, and an Organization Chart Manager (OCM). These have the following roles:

- A query manager agent receives a query from a user, develops an initial high-level information gathering plan, recruits agents from the necessary FUs via the FMs to form a Query Answering Unit (QAU) and monitors plan execution.
- A functional unit is a collection of agents which have access to a particular type of information resources. Each functional unit has a functional manager. Upon a request from a query-answering agent, the FM agent assigns tasks to agents within its FU. Although meant for the same type of information sources, agents within a specific FU may have different expertise. The FM takes these differences into account when planning task assignment.

- Low-level information agents are simple agents, each specialized in a specific type of information retrieval. Such an agent, upon a request for information from a functional agent, goes out to the web and retrieves it. The information agent is not assumed to plan the retrieval or manipulate the resulting information. These activities are performed by functional agents.
- The organization chart manager is an organizational memory where agents can look for information regarding the availability and the capabilities of other agents. New agents are added to the OCM when they join the system (dynamically, during run time). QMs, FMs and functional agents all consult the OCM to locate the appropriate agents to which they have to delegate tasks. Note that while supporting dynamic changes in the agent community, the OCM is a centralized mechanism and a single point of failure.

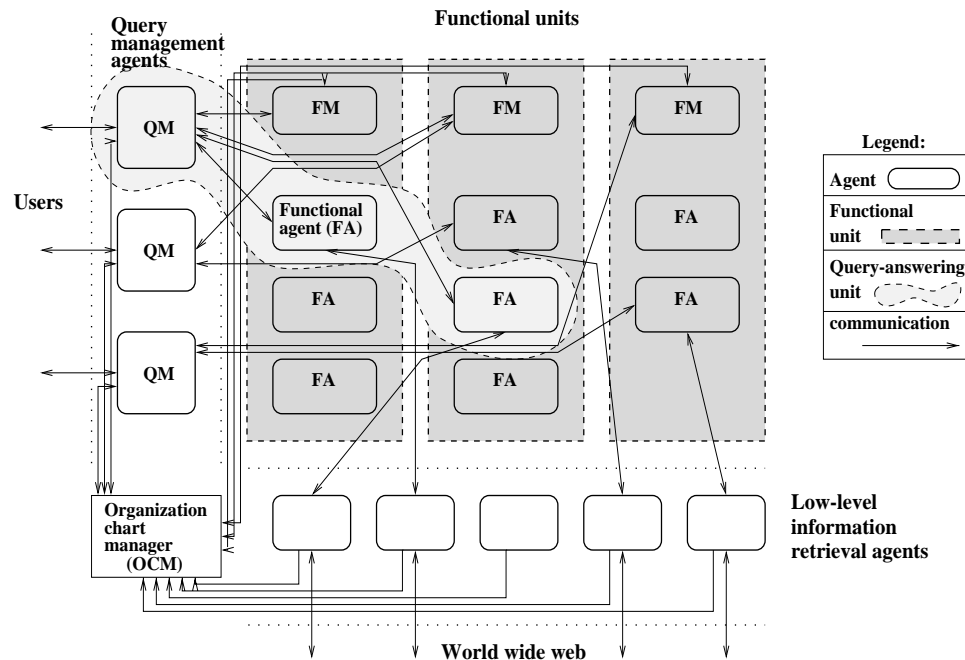


Figure 8: The MACRON multi-agent organization.

In figure 8 the components of MACRON and the relationships between them are presented. Some details were omitted since the figure is already dense without them. The bi-directional communication of functional agents (FA) with the OCM does not appear in the figure however exists in the system. In fact, all of the agents communicate with the OCM. The exception is the low-level information-retrieval agents, who only provide the OCM information about themselves when they join the system, but do not use the OCM for finding other agents. It should also be noted that we have marked (using a dashed curved closed line) only one of the query answering units, although there are three QAU in the figure. Another connection that does not appear in the figure is the communication between each FM and the FAs in its functional unit. Since FMs assign tasks to FAs, such communication exists. Of course, once the QAU was formed, the monitoring of the FAs is done by the QM responsible for the query and not by the FM.

The MACRON organization is referred to as a matrix organization since it has a two-dimensional partition to units: functional units and query-answering units. This indeed increases the flexibility

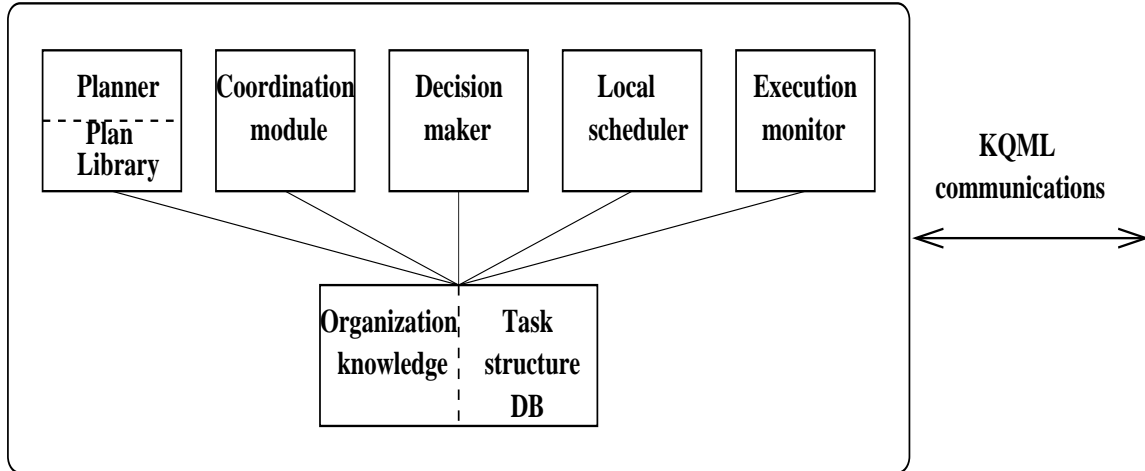


Figure 9: DECAF – the MACRON internal agent architecture.

of the systems to create team of agents dynamically. It also facilitates specialization. On the other hand, each functional unit has a single FM that connects FA to QAUs, and finding agents is possible only via the OCM. Both of these are single points of failure, and the latter is completely centralized, which is usually an undesirable property in MAS. However, the OCM provides MACRON with support for dynamic MAS openness, since agents can dynamically register at the OCM during system execution.

3.5.1 DECAF – the MACRON Internal Agent Architecture

The internal architecture of agents in MACRON is called DECAF (Distributed Environment Centered Agent Framework). This framework is described in figure 9. The components of an agent and the interactions among them are as follows:

- The organization knowledge and the task structure DB are a memory structure accessible to all of the other components of the agent.
- The planner, which usually has a domain specific plan library, create task structures based on incoming queries or tasks and the plan library.
- The coordination module checks the task structure DB for patterns of interdependencies and constraints, and suggests new tasks to resolve these.
- The scheduler create several schedules for tasks in the task structure DB to be executed.
- The decision maker selects an appropriate schedule according to environmental and performance criteria.
- The execution monitor monitors the execution of actions from the current schedule.

The functional modules do not interact directly with one another. They all approach the shared memory structure, thus they do not have mutual interface dependencies. This modularity allows

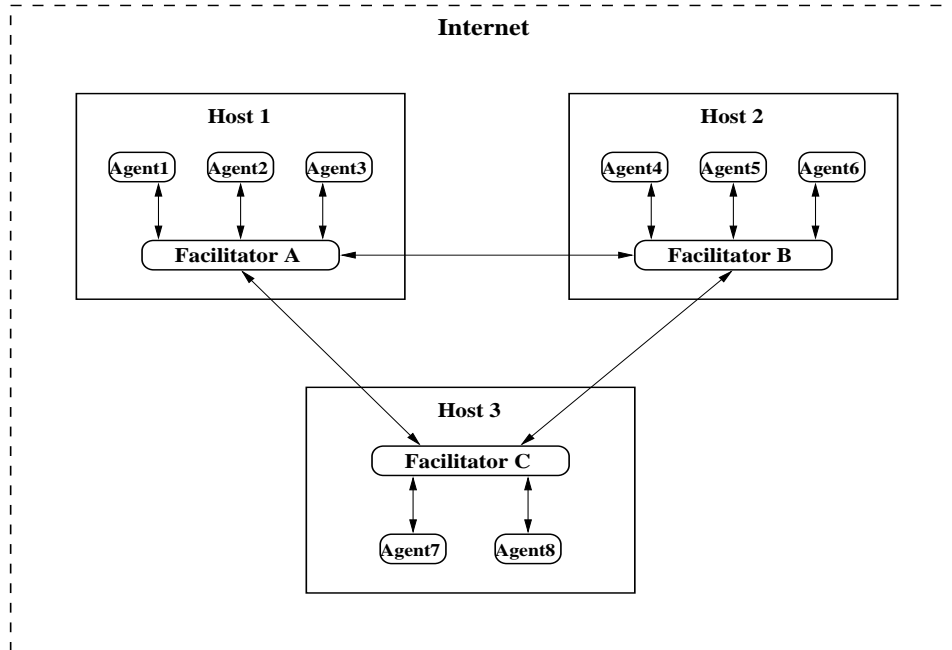


Figure 10: The federated multi-agent organization.

for replacing modules with no changes to the other modules (e.g., if one wants to introduce a new, better scheduling mechanism, it is only necessary to replace the scheduler, and other modules are not affected).

3.6 Federated MAS

An different approach to constructing multi-agent systems is presented in [7]. There, Genesereth discusses the need for programs to coordinate and inter-operate, and introduces a system organization which consists of agents and *facilitators* as a means for achieving this inter-operability. Facilitators and agents are typically organized into a federated system as illustrated in figure 10 (an example of such a system is OAA [17]). The federated organization suggests that agents do not communicate directly with each other. Instead, each agent communicates with its local facilitator and facilitators, in turn, communicate with one another. In this way each group of agents who are facilitated by a single facilitator is a *federation* in which an agent surrenders some of its autonomy to the facilitator. The number of facilitators is not bound, they may be running on multiple machines, and the network topology between them is arbitrary. In this organization, messages which are sent to facilitators are not addressed to specific agents. It is the facilitator's role to direct messages to the agents that can handle them.

The federated organization provides coordination among agents based on a specification-sharing approach to inter-operation. Agents can dynamically connect and disconnect from a facilitator. Within each federation, each agent agrees to service requests sent to it by its facilitator, and in return the facilitator handles the requests posted by the agent. Upon connection to a facilitator, an agent provides a specification of its capabilities and needs in an agent communication language (ACL). An agent also sends to its facilitator application-level information and requests and receives

such information and requests in return. A facilitator transforms the application-level messages and route them to appropriate agents. Note that this transformation may be very sophisticated. It may require decomposition of a message into several messages to be sent to several recipients, or bundling several messages together. It may also require translation of messages from their original form to a form understood by their recipients.

The federated organization facilitates application inter-operability, however compromises agent autonomy. Inter-operability is addressed in MAS research where, instead of facilitators, other types of middle agents [4] are used (e.g. matchmakers in [32]). Note that inter-operability is supported by the other multi-agent infrastructures presented above as well, however in a less flexible and dynamic manner. Another advantage of facilitators is the (dynamic) openness of the system, which allows for connecting and disconnecting agents dynamically. Federated MAS which use facilitators allow, at least in concept, the inclusion of agents of heterogeneous architectures in a single system. This will require, however, rather sophisticated facilitators which, to our best knowledge, are not yet available. The main drawbacks of federated MAS is the need to surrender agent autonomy, which may violate the privacy and harm the interests of the agents involved, and the need to design and implement sufficiently sophisticated (and unbiased) facilitators.

3.7 The RETSINA Multi-Agent Infrastructure

RETSINA (Reusable Task Structure based Intelligent Network Agents) [32] is a multi-agent infrastructure that was developed to integrate information gathering from web-based sources and perform decision support tasks. RETSINA includes a distributed MAS organization, protocols for inter-agent interaction and collaboration, and a reusable set of software components for constructing agents. It consists of three types of agents (see Figure 11). *Interface agents* interact with the user receiving user specifications and delivering results. *Task agents* support task performance by formulating problem solving plans and carrying out these plans. *Information agents* provide intelligent access to a heterogeneous collection of information sources. Tasks that cannot be executed by a single agent are performed by a team, and such teams are formed on demand. A similar infrastructure is used in InfoSleuth, and the communication protocol used is similar as well [20].

In open MAS, agents must be able to locate one another. In such systems which are distributed over the Internet, where participating agents may dynamically enter and leave, which is distributed over the Internet, broadcast communication solutions are precluded. In RETSINA this problem is solved by introducing matchmaking agents [16, 4]. The process of matchmaking allows an agent A with some tasks, to learn the contact information and capabilities of another agent, B , who may be able to execute part of A 's tasks via a matchmaker which is an agent that maintains the contact information of other agents. Agents that join the system advertise themselves and their capabilities to a matchmaker, and when they leave the agent society, they un-advertise. In search of other agents, agents approach a matchmaker and ask for names of relevant agents. After having acquired the information about other agents they can directly contact these agents and initiate cooperation as needed. To relax the problem of unavailable or overwhelmed single matchmaker, RETSINA provides a protocol for information coherence among multiple matchmaker agents. By these means RETSINA supports dynamics openness.

The agents in RETSINA compartmentalize specialized task knowledge, organize themselves to avoid processing bottlenecks, and can be constructed specifically to deal with dynamic changes

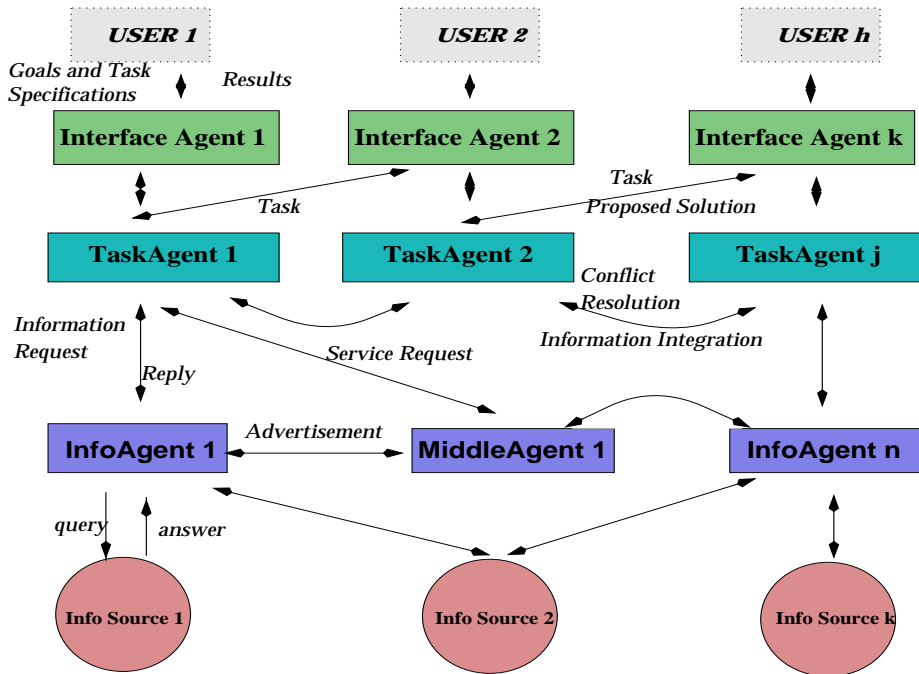


Figure 11: The RETSINA multi-agent infrastructure

in information, tasks, number of agents and their capabilities. The agents are distributed and run across different machines. Based on models of users, agents and tasks, the agents decide how to decompose tasks and whether to pass them to others, what information is needed at each decision point, and when to cooperate with other agents. The agents communicate with each other to delegate tasks, request or provide information, find information sources, filter or integrate information, and negotiate to resolve inconsistencies in information and task models. The communication language used is KQML.

3.7.1 The RETSINA Agent Internal Architecture

The RETSINA agent architecture is based on a multi-module, multi-thread design. It consists of two types of components: functional units and data storages. A RETSINA agent uses four data storage modules, as follows.

- The *objective database* is a dynamic storage. It stores the objectives of the agent of which it is a component. An objective DB implements a queue, i.e., the first objective on the queue is handled first by the *planner*. New objectives are inserted to the queue by the *communicator* (from outside sources) and by the planner (from inside sources), as a result of (partial) planning. As a result of planning new objectives may be created, or some decomposed tasks are still in the level of abstraction which is considered an objective.
- The *task database* is a dynamic storage. It stores tasks which were reduced to the lower level, i.e., to actions. These tasks may still be not ready for execution and wait in the task DB until the required conditions for their execution are set true (via outcome propagation).

When this happens, the actions are considered enabled, and are scheduled for execution by the *scheduler*.

- The *task schema library* is a static data storage that holds tasks schemas. These are used by the planner for task instantiation. The task schema library is created off-line by the designers of RETSINA agent system. Some of the schemata are generic and apply for any problem domain, whereas others are domain specific schemata.
- The *task reduction library* is a static data storage that holds reductions of tasks. These are used by the planner for task decomposition. The task reduction library created off-line and consists of generic as well as domain-specific data items.

The functional modules of the RETSINA agent, each of them an autonomous thread of control, use the data storages as follows:

- The communicator receives and sends messages, parses incoming messages and creates objectives which are inserted to the objective DB.
- The planner performs instantiation and reduction of tasks. It takes the objectives of its agent, decomposes them to lower level tasks, and the tasks which are executable (referred to as actions) are passed forward, to be scheduled for execution. The execution of the actions may affect some of the tasks which are still in the planning phase, by outcome propagation.
- The scheduling of enabled tasks is performed by the *scheduler*, which is an autonomous thread of control as well. It takes enabled actions from the task DB and puts them, scheduled, in the *schedule* (which is a dynamic data storage).
- Activation of actions in the schedule is performed by the *execution monitor* thread. For each action on the schedule it creates a separate thread of control, and it monitors the activity of each of these working threads. Action threads may, during their execution or at termination, propagate outcomes to either the execution monitor or to any objective, task or action in all of the dynamic data storages.

The modules described above and the connections between them are depicted in figure 12.

The modularity of the RETSINA agent architecture and having no direct interfaces between its functional modules results in code re-usability (e.g., the RETSINA communicator is used for multiple type of agents as well as non-agent applications that need to converse with agents). In addition, functional components can be easily replaced in a plug-in fashion (e.g., if one would like to introduce an improved scheduling mechanism, only the scheduler should be replaced, and this requires no changes in the other functional modules).

To summarize, the RETSINA infrastructure, supports flexible, dynamic organization (based on a flat organization) in an open environment. It also supports code re-usability and robustness and agent asynchrony. However, it introduces some code redundancy. This is an inevitable result of its adaptability to dynamic changes in tasks, agents and the working environment, which require duplicate expertise as well as multiple middle agents. The dynamic organization of RETSINA results in computation and communication overheads as well as additional code for the formation of teams on demand. Less flexible system organizations avoid this overhead (however they are less adaptable to changes).

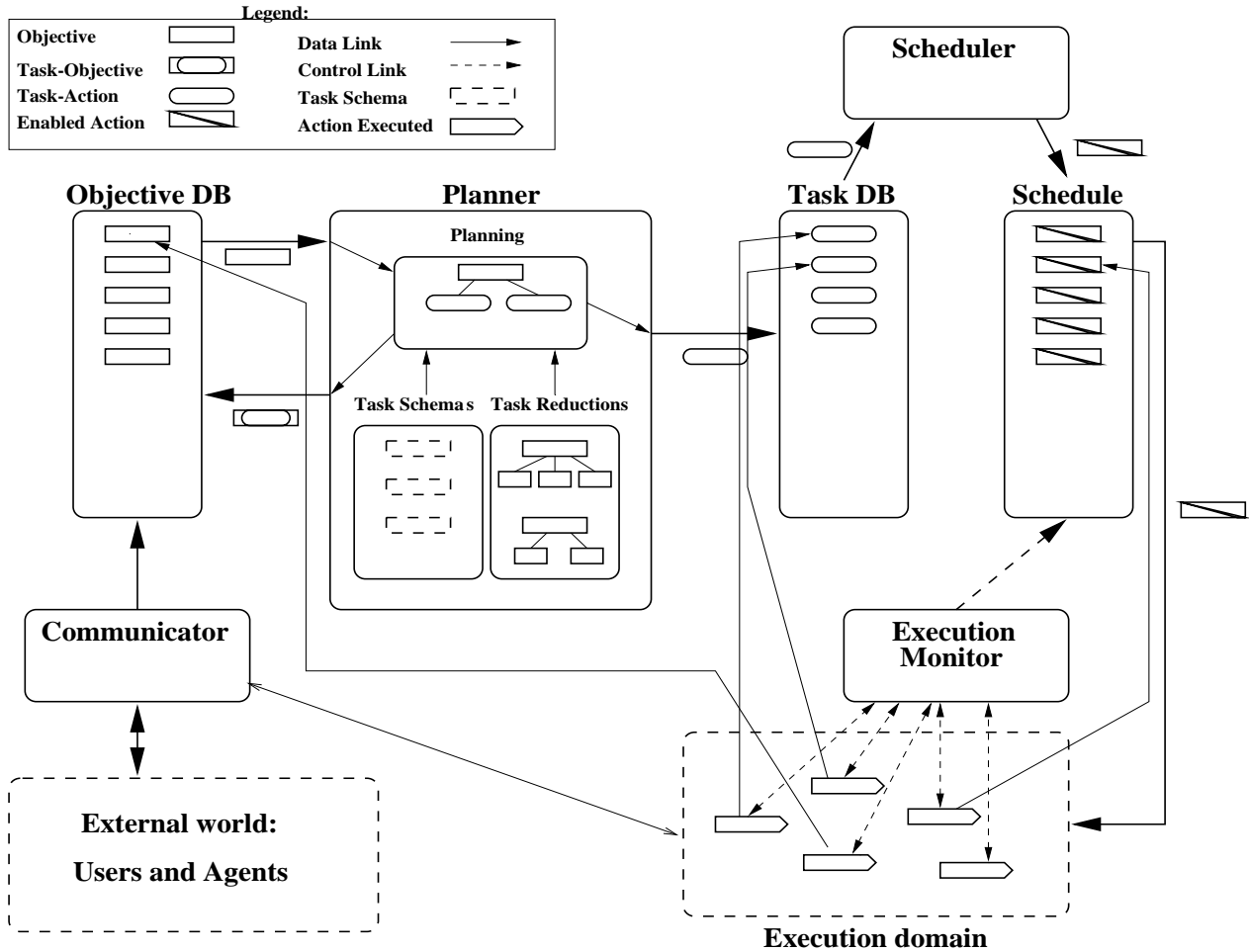


Figure 12: The RETSINA agent architecture.

4 Concluding Remarks

In this research we have made a first step in the direction of analyzing multi-agent systems as a software architecture style. Pursuing this direction shall make MAS more accessible to system designers and provide them with means for considering MAS as one of the prospective solutions to computational problems.

As we have shown throughout the study, MAS are a unique software architecture, distinguishable from other architectures, which provides solutions to a specific family of computational problems. Yet, further investigation of the properties of these systems is still necessary. Among others issues, efficiency properties of MAS, and in particular a rigorous comparison between MAS and other software architecture styles, were scarcely examined. Inter-operability, too, is yet to be investigated to provide the appropriate mechanisms for its implementation. Such additional research should allow for more adequate match between MAS and the problems they solve. In addition, it is necessary to be able to compare between different MAS and the solutions they provide to a given problem. One would also seek development tools and generic agent-oriented

programming languages⁵ move from the labs to the field. If multi-agent systems are to succeed as a software architecture style, design methodologies, development tools, programming languages and evaluation criteria should become an inseparable part of this paradigm. In our work, we merely address design issues and evaluation criteria. Even here, we leave many open questions and a large amount of work to be performed in future research.

⁵Initial work in this direction can be found in, e.g., [18, 30].

5 Acknowledgement

I would like to thank Mr. Anandeeep Pannu for his insightful remarks upon which I was able to significantly improve the content of this report.

This research could not have been performed without the support of the software agents research group at the Robotics Institute.

References

- [1] Joseph Bates. The role of emotion in believable agents. *Communications of the ACM, Special Issue on Intelligent Agents*, 37(7):122–125, July 1994.
- [2] Frances Brazier, Barbara Dunin-Keplicz, Nick R. Jennings, and Jan Treur. Formal specification of multi-agent systems: a real-world case. In Victor Lesser, editor, *Proceedings of the First International Conference on Multi-Agent Systems*, pages 25–32, San Francisco, CA, 1995. MIT Press.
- [3] C. Castelfranchi. Social power. In Y. Demazeau and J. P. Muller, editors, *Decentralized A.I.*, pages 49–62. Elsevier Science Publishers, 1990.
- [4] K. Decker, K. Sycara, and M. Williamson. Middle-agents for the internet. In *Proceeding of IJCAI-97*, pages 578–583, Nagoya, Japan, 1997.
- [5] Keith Decker, Victor Lesser, M. V. Nagendra Prasad, and Thomas Wagner. MACRON: An architecture for multi-agent cooperative information gathering. In Tim Finin and James Mayfield, editors, *Proceedings of the CIKM '95 Workshop on Intelligent Information Agents*, Baltimore, Maryland, 1995.
- [6] Tim Finin, Rich Fritzon, Don McKay, and Robin McEntire. KQML – A language and protocol for knowledge and information exchange. In *Proceedings of the 13th International Workshop on Distributed Artificial Intelligence*, pages 126–136, Seattle, WA, July 1994.
- [7] Michael R. Genesereth and Steven P. Ketchpel. Software agents. *Communications of the ACM, Special Issue on Intelligent Agents*, 37(7):48–53, July 1994.
- [8] Robert S. Gray, David Kotz, George Cybenko, and Daniela Rus. D’agents: Security in a multiple-language, mobile-agent system. In Giovanni Vigna, editor, *Mobile Agent Security*, Lecture Notes in Computer Science. Springer-Verlag, 1998.
- [9] Robert S. Gray, David Kotz, Saurab Nog, Daniela Rus, and George Cybenko. Mobile agents for mobile computing. Technical Report PCS-TR96-285, Dartmouth College, Computer Science, Hanover, NH, May 1996.
- [10] J. Y. Halpern. A theory of knowledge and ignorance for many agents. Technical Report RJ 9894, IBM Research Division, 1994.
- [11] Thomas Haynes and Sandip Sen. Learning cases to resolve conflicts and improve group behavior. *International Journal of Human-Computer Studies (IJHCS)*, 1997. (accepted for publication).
- [12] M. Huhns and M. Singh (editors). *Readings in agents*. Morgan Kaufmann, 1998.
- [13] N. Jennings and M. Wooldridge (editors). *Agent technology*. Springer, 1998.
- [14] S. Jha, P. Chalasani, O. Shehory, and K. Sycara. A formal treatment of distributed matchmaking. In *Proceeding of Agents-98*, pages 457–458, Minneapolis, Minnesota, 1998.

- [15] S. Kraus, J. Wilkenfeld, and G. Zlotkin. Multiagent negotiation under time constraints. *Artificial Intelligence*, 75(2):297–345, 1995.
- [16] D. Kuokka and L. Harada. On using KQML for matchmaking. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 239–245, San Francisco, June 1995. AAAI Press.
- [17] D. L. Martin, A. J. Cheyer, and D. B. Moran. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 1999.
- [18] F. McCabe and K. Clarck. April – agent process interaction language. In M. Wooldridge and N. Jennings, editors, *Intelligent Agents*, Lecture Notes in Artificial Intelligence No. 890, pages 324–340. Berlin:Springer-Verlag, 1995.
- [19] T. J. Mowbray and W. A. Ruh. *Inside CORBA - Distributed object standards and applications*. Addison Wesley, 1997.
- [20] M. Nodine and A. Unruh. Facilitating open communication in agent systems: the infosleuth infrastructure. In M. Singh, A. Rao, and M. Wooldridge, editors, *Intelligent Agents 4*, Lecture Notes in Artificial Intelligence No. 1365, pages 281–296. Berlin:Springer-Verlag, 1997.
- [21] T. Norman, N. Jennings, P. Faratin, and E. Mamdani. Designing and implementing a multi-agent architecture for business process management. In J. Muller, N. Jennings, and M. Wooldridge, editors, *Intelligent Agents 3*, Lecture Notes in Artificial Intelligence No. 1193, pages 261–275. Springer-Verlag, 1996.
- [22] H. Nwana and N. Azarmi (editors). *Lecture notes in artificial intelligence vol. 1198: Software agents and soft computing*. Springer, 1997.
- [23] Anand S. Rao and Michael P. Georgeff. BDI agents: from theory to practice. In Victor Lesser, editor, *Proceedings of the First International Conference on Multi-Agent Systems*, pages 312–319, San Francisco, CA, 1995. MIT Press.
- [24] J. S. Rosenschein and G. Zlotkin. *Rules of Encounter: Designing Conventions for Automated Negotiation Among Computers*. MIT Press, Boston, 1994.
- [25] T. W. Sandholm and V. R. Lesser. Coalitions among computationally bounded agents. *Artificial Intelligence*, 94:99–137, 1997.
- [26] E. Scalabrin and J. P. Barthes. Osaca : une architecture ouverte d’agents cognitifs independants. In *Actes de la Journee "Systemes multi-agents"*, Montpellier, France, 1993.
- [27] M. Shaw and D. Garlan. *Software architecture : perspectives on an emerging discipline*. Prentice Hall, New Jersey, 1996.
- [28] O. Shehory and S. Kraus. A kernel-oriented model for coalition-formation in general environments: Implementation and results. In *Proc. of AAAI-96*, pages 134–140, Portland, Oregon, 1996.

- [29] Weiming Shen and Jean-Paul Barthes. DIDE: A multi-agent environment for engineering design. In Victor Lesser, editor, *Proceedings of the First International Conference on Multi-Agent Systems*, pages 344–351, San Francisco, CA, 1995. MIT Press.
- [30] Y. Shoham. Agent oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.
- [31] Y. Shoham and M. Tennenholtz. On the synthesis of useful social laws for artificial agent societies. In *Proc. of AAAI-92*, pages 276–281, California, 1992.
- [32] K. Sycara, K. Decker, A. Pannu, M. Williamson, and D. Zeng. Distributed intelligent agents. *IEEE Expert – Intelligent Systems and Their Applications*, 11(6):36–45, 1996.
- [33] A. S. Tanenbaum. *Computer networks*. Prentice Hall, 1988.
- [34] T. Wittig, editor. *ARCHON: an architecture for multi-agent systems*. Ellis Horwood, 1992.
- [35] M. Wooldridge. Agent-based software engineering. *IEE Proceedings on Software Engineering*, 144(1):26–37, 1997.