Organizational Principles for Multi-Agent Architectures

van Aart, C. J.

Citation for published version (APA):
Interoperability within a Complex Multi-Agent Architecture

This chapter is partly based on two deliverables of the IBROW project: D15 Brokering in IBROW and D10 Interoperability co-authored by B.J. Wielinga, A. Anjewierden and W. Jansweijer. The goal of the IBROW project (Intelligent Brokering on the Web, see http://ibrow.swi.psy.uva.nl) is to develop technologies for (semi-)automatic selection and configuration of new applications by reuse of existing services. Work on a multi-agent architecture capable of (semi-)automatic reuse of Problem-Solving Methods (PSMs) is discussed. Using the notion of separation of concerns, specialized agents are defined that operate within virtual environments. The agents within the architecture collaborate using specialized ontologies and collaboration patterns on top of an interoperability structure. A proof of concept is presented that explains the dynamics of parts of the architecture.

5.1 Introduction

This work addresses the problem of interoperability within a distributed architecture composed of heterogeneous components. The architecture supports the composition of applications from existing (web) services that reside on the Web. These (web) services range from simple information retrieval from databases to knowledge-based consultation services. Such services can be seen as problem-solving methods (PSMs) for knowledge-based systems (KBSs) [Schreiber et al., 1999].

Most existing (web) services are distributed, heterogeneous and rigid, in the sense that they can not easily be configured. For example, the Semantic Web community has developed the view that it is unrealistic to assume that the content producers conform to a single standard and ontology. Different languages will be used for competence representations. Some service providers will use SOAP and WSDL as technical competence specification mechanism, others will use RDF(S) and OWL as ontology representation languages. The libraries of services (i.e. PSMs) of the future Semantic Web will be as heterogeneous as the current collection of search engines and other services that exist on the Web. Therefore, we need an approach that takes heterogeneity and distribution into account.

One way to deal with distributed and heterogeneous services is to apply intelligent software agents. Agents provide a natural way to describe distributed heterogeneous services [Genesereth, 1997]. For example, every service can be seen as an individual agent. Moreover, agent technology provides supporting technology and standards, including communication platforms, message transport mechanisms, message formats and white and yellow page services. Furthermore, the agent concept provides a metaphor to reason about processes and coordination [Bond and Gasser, 1988].

1A web service is a software process that can be invoked remotely using web technology.
In our view, (web) services can conceptually be represented by agents. In order to exchange information between different services, agent wrappers can be built, so that these services can be unlatched to other services. The idea is that services can consult other services by using agents as intermediate. The rationale behind using agents is that they put an additional layer on existing services in order to have a common means for communication and coordination [Genesereth, 1997]. However, interoperability problems such as different communication languages, datamodels, infrastructures and coordination mechanisms still have to be solved.

Although web services and agents are already deployed in various domains, many of them tend to be inflexible: it is not possible to modify the underlying system, neither configure them for other domains, nor to integrate different services to produce new functionalities. Furthermore, most web services are heterogeneous and not designed to interact with other services. For example, Google provides a SOAP interface to its search services, which can be invoked from within an application\(^2\). However, the interface itself is not a service, it still needs to be embedded into another service.

In this chapter, we present an architecture that enables cooperation among agents capable of configuring and executing new applications composed of existing services present on the Web. Such an architecture could change the nature of using software from a centralistic compositional approach to a distributed (agent-based) plug and play process. The focus is on enabling interoperability among the agents that represent heterogeneous and distributed services.

This chapter discusses the outcomes of the IBROW project at several levels of detail. In Section 5.2 we discuss the IBROW approach in general. The Agent Architecture that supports the IBROW approach is presented in Section 5.3. In order to have the agents within the IBROW architecture interoperate, we outline the Interoperability Framework in Section 5.4. Section 5.4 discusses how we implemented the IBROW system. Finally, we present a proof of concept that explains the dynamics of parts of the IBROW system in Section 5.6.

5.2 IBROW Approach

Reuse of knowledge and knowledge-based software components has always been an important goal of the knowledge engineering community. With the explosive growth of the Web, new opportunities for reuse arise: knowledge-intensive services and components can be offered on the Web. There are various PSMs, web services and resources available on the Web that could be linked together to form new applications. Several PSM libraries with corresponding operational components are now available [Fensel and Motta, 2001, Benjamins, 1993, Eriksson et al., 1995, Motta, 1999, Motta and Lu, 2000].

The goal of the IBROW project is to develop technologies for (semi-)automatic selection and configuration of PSMs\(^3\) in order to compose new applications. The idea is that users interact with a service, specifying the task that an application should perform (i.e. goal specification). This service is called the broker, i.e. a service that mediates among demanding parties, such as users, and offering parties, such as PSM providers. Subsequently, the broker searches for PSMs on the Web and - if successful - configures an application that will solve the user's task [Benjamins et al., 1998].

In order to explain the IBROW approach, we divide its functionality into a number of spaces. A space is a virtual environment that clusters processes (such as agents and PSMs) and resources (such as libraries) that are distributed on the Web. Three spaces are defined: the user space, the

\(^2\)www.google.com/apis/

\(^3\)In the remainder, we use a very general notion of PSMs, consequently we see (web) services as PSMs.
broker space and the execution space, see Figure 5.1. The user interacts with the user space in order to formulate a goal for an application. A goal can be specified using the Universal Problem Modeling Language (UPML) framework [Fensel et al., 1999b].

The broker space is able to mediate between the user space and the many PSMs available on the Web, potentially capable of realizing the user's goal. The PSMs are organized in specialized PSM libraries, which provide competence descriptions to the broker space on request. These libraries are not part of the IBROW architecture, because these are offered by third parties. Therefore we did not define a library space. On the basis of the goal of the user, the broker space can configure custom-made applications using existing (heterogeneous and distributed) PSMs. For further reading on a centralized PSM broker we refer to [Benjamins et al., 1998].

The execution space is able to execute applications based on the output of the brokering process: the application configuration. Execution of applications involves invocation of PSMs and coordination over invocations of PSMs. The sequence of invocation and the coordination over it, is defined in a MAP (Multi-Agent Plan). This plan also handles the input/output mappings between the PSMs involved.

The problem now is that users, libraries and PSMs are distributed on the Web. Furthermore, the user, the broker and the execution spaces are too complex to build in one system. Therefore, we further separate the functions in each of the spaces into specialized agents.

5.3 Agent Architecture

In this section, we elaborate on the spaces defined above and define specialized agents that operate within the spaces. The idea is that the agents will not be integrated into one system. Rather the agents will be organized in a multi-agent architecture, see Figure 5.2.
The use of agents comes with a number of advantages. First, agents are capable of coupling distributed processes, without centralizing control. Using wrapping technology, an agent can form an interface between distributed processes, such as PSMs, and other agents [Genesereth and Ketchpel, 1994]. Secondly, specialized or generalized agents can easily replace existing agents. The reason for this is that agents do not share a common memory or common libraries of functions [Wooldridge, 2002]. Therefore, the replacement of one agent does not affect other agents. Finally, the control over the overall architecture is distributed. Every agent is responsible for a part of the overall functionality, because the required integration knowledge can also be distributed. For that reason, the agent is partially independent of other agents (i.e. autonomous). The remaining dependency between agents can be handled by coordination mechanisms where communication and flow of information is regulated.

We first discuss the agents that operate within the spaces. Next interoperation is discussed.

5.3.1 User Space

The end-users of the IBROW system interact with the user space. Since users are distributed on the Web, user agents are allocated to individual users. The user agent represents an end-user and hides the complexity of the overall system. Several variations on the user agent related to the user’s level of expertise are possible, ranging from novice to expert user agents.

Figure 5.2
The IBROW multi-agent architecture showing the agents that operate within the user, broker and execution space. The numbered lines (① - ⑥) represent collaborations between agents, which are described in Table 5.3 (p.95).
In order to acquire the goal specification and domain knowledge from the user, the user agent uses an interface. For details on interfaces of the IBROW system we refer to [Wielinga et al., 2003]. Furthermore, it passes goals to the agents in the broker space and input to the agents in the execution space. The goal descriptions are in terms of input and output roles, competence descriptions in terms of pre- and post-conditions, and domain ontologies. The goal descriptions can be specified within the UPML framework. In the remainder, we do not discuss UPML in full detail. Details on UPML, including the UPML meta-model can be found in [Fensel et al., 1999b, Omelayenko et al., 2000].

Finally, the user agent presents results received from the agents in the execution space to the end-user.

5.3.2 Broker Space

PSM libraries and the actual broker process are located within the broker space. In order to locate PSMs, PSMs are clustered in specialized libraries. Every library agent represents one PSM library and provides PSM descriptions, expressed in UPML, to the broker on request. Library builders maintain the PSM libraries.

The broker space is responsible for the following tasks:

* **Maintain interaction with the user space.** The user space delegates goals of users to the broker space.

* **Retrieve and select competence descriptions of suitable PSM candidates.** PSM descriptions (in UPML) will be retrieved from PSM libraries, in order to select the appropriate PSMs to accomplish the user's goal.

* **Configure and adapt the selected PSMs.** Based on the selected PSMs, an application configuration is compiled. This configuration explains how the selected PSMs need to be configured. Possible configurations are the location of input and the required knowledge bases.

* **Delegate application configuration to the execution space.** The broker space does not take care of the actual execution of the configuration. Therefore, it sends the application configuration to the execution space.

* **Inspect the outcome of execution space and reconfigure the application configuration.** When the execution space has executed the application configuration, it reports to the broker space. Based on reconfiguration strategies (such as the Propose Critique Modify (PCM) algorithm), the broker space reconfigures the configuration (if necessary) and delegates it to the execution space.

The broker tasks can be with different levels of support to the user, ranging from giving interactive assistance in manual selection and configuration of PSMs, to the fully-automatic configuration of an intelligent problem solver. For example, manual selection and configuration of PSMs can be done with the Internet Reasoning Service\(^4\), which supports the semi-automatic configuration of knowledge-based applications on the Web.

We have defined two types of brokers: the Static Broker agent and the dynamic broker agent: the Reconfigurator. The static broker agent defines the initial application configuration. After

\(^4\)http://kmi.open.ac.uk/projects/irs
receiving a goal specification from the user agent, it contacts several library agents for PSM selection. From there, it constructs an application expressed in an application configuration. The application configuration is then delegated to the execution space.

The Reconfigurator helps the agents in the execution space to refine the configuration of the broker. This works as follows; the execution space reports (Propose) the output of an application configuration along with the application configuration itself to the Reconfigurator. The Reconfigurator evaluates (Critique) the output based on a set of criteria. The result of the evaluation is an altered (Modify) application configuration. The modified application configuration will be sent to the execution space. This process repeats until the set of criteria is satisfied. Examples of criteria are the quality and the quantity of the result set. When the Reconfigurator instructs a new configuration, the agents within the execution space take care of the execution. Otherwise, in case of acceptance, the agents within the execution space report the result set to the user agent.

5.3.3 Execution Space

The agents in the execution space are Operators and Managers. These two agent roles are based on the Operator and Manager roles as introduced in Section 2.2. Operators represent PSMs and are able to configure and invoke PSMs. The Manager is responsible for coordinating the Operators. The reason to use two agent types for the execution of an application configuration, is to separate the knowledge for invocation of the PSMs from knowledge for the coordination over the invocations of the PSMs. The advantage of this approach is that there is not one single complex agent responsible for the execution phase, rather a collection of specialized agents. Every agent can choose on an individual basis how to perform its activities in order to achieve its goals. This point is discussed in more detail below.

The main tasks of the execution space are:

*Translate the application configuration produced by the broker, into a MAP.* For every selected PSM, a PSM provider is selected.

*Select a coordination strategy.* A coordination strategy provides structures to follow the control structure of the application configuration (see also Chapter 3). Furthermore, it regulates the flow of inputs and outputs between PSMs. The coordination strategy is integrated in the MAP.

*Negotiate with PSM providers.* On the basis of the MAP, negotiation with PSM providers is initiated. A negotiation involves the configuration of a PSM. The idea is that PSM providers (i.e., agents) are relatively independent and have shielded off the functionality of the PSM, in such a way that PSMs cannot be directly invoked. Therefore, an explicit session with the PSM providers has to be started.

*Execute the Multi-Agent Plan (MAP).* Based on the steps in the MAP, the involved PSM providers are contacted in order to invoke PSMs. The flow of input objects and output objects is regulated according to the MAP.

*Handle exceptions or failures raised by PSM providers or by loss of information.* The execution environment takes appropriate steps in case something goes wrong. Possible situations of this type are a PSM provider that does not respond and unprocessable information.

*Report the results of an execution to the Reconfigurator.* The execution space waits for the reaction of the Reconfigurator.
In order to construct PSM providers we look at the problems involved in transforming a PSM into an agent, in such a way that it can be coupled to other PSMs providers and Managers. First, most PSMs are not meant to be on the Web, they form a part of a larger (centrally controlled) set of KBSs. Secondly, PSMs do not address issues like communication, session management, multi-user support and web standards. Thirdly, the interface to a PSM, i.e. the way to configure it and invoke it is not always specified or clear. Fourthly, more advanced PSMs need configuration before they can be invoked. Finally, there are complex PSMs that require interaction with the user or other systems.

As a solution, only one agent type could be defined that is able to invoke all PSMs defined in the application configuration. However, this is not possible because PSMs are distributed on the Web and PSMs are heterogeneous. Therefore, we need an addressing mechanism, in order to invoke the PSMs. Furthermore, there should be a transport mechanism to establish a transaction with the PSM. Examples of transport mechanisms are TCP/IP and IIOP. Secondly, the PSMs are heterogeneous, which means that for every PSM a separate invocation mechanism has to be defined. An invocation mechanism can be seen as a communication protocol above a transport mechanism, such as HTTP, SOAP or CORBA. Finally, the PSMs behave differently, hence for every PSM a separate transaction mechanism has to be defined. Transaction mechanisms deal with how to encode and decode information from one format to another. Examples of information transactions are XML to plain text, RDF to SQL and so forth.

Another solution is to introduce a mediator, which is able to operate an individual PSM and can communicate with other mediators. An (IBROW) Operator is a type of mediator that translates agent communication to proprietary instructions. In order to contact a PSM, a transducer can be applied [Genesereth and Ketchpel, 1994]. A transducer can map instructions from an agent to a service and vice versa. It can also map in the reverse sense, that is, instructions from service to agent and results from agent to service. The difference between the two mappings is that the former is part of a reactive behavior, the latter is part of a pro-active behavior.

This approach has the advantage that the agent does not require knowledge of the configuration and invocation of these services. The discussion on the use of annotation languages, such as WSDL, DAML and OWL, and the use of other deployment techniques for PSMs is not part of this work.

Given the fact that PSM invocation is complex and that for every PSM a separate Operator has to be defined, we separated the actual invocation of PSMs, i.e. PSM consultation, and the coordination over the PSM invocations. Invocation of PSMs is handled by Operators. Coordination is handled by the Manager. The Manager coordinates the execution of the application configuration by constructing a MAP. This MAP defines the sequence used to consult Operators that are able to invoke the required PSMs. The next step for the Manager is to start a negotiation with the involved Operator on the PSM configuration and the Operator's role in the MAP. The Manager starts the actual execution by consulting Operators. After the execution of the MAP, the Manager reports its result to the Reconfigurator.

The idea behind an Operator is that it provides an interface to a PSM. This interface is written in such a way that a PSM can be consulted as-is. This means that the Operator takes care of transport of information from and to a PSM, using the appropriate protocols and transactions. Deployment of PSMs can be done in two steps: (1) deployment of a competence description using UPML via PSM libraries or (2) deployment of PSM invocation via mediators (called Operators) that have access to PSMs.

These issues also lead to the conclusion that it is not trivial to define a single API (as in SOAP) for PSMs that can be placed above PSMs.
The competences and states of the Manager are summarized in a 5C model (according to Section 4.2 (p.66)) agent design, see Table 5.1 (p.92). As shown, the communication model interacts with other agents in the IBROW system on the basis of the available ontologies. These ontologies are discussed in Section 5.4.3 (p.94). The competence model is able to execute four main tasks: MAP construction, Operator negotiation, operate MAS and negotiate with Reconfigurator. The execution of these tasks for a specific domain is discussed in Section 5.6.2. The management of the agent's life cycles (cf. Section 3.4) is handled by the self model, which instructs the planner model. The specific states and transitions are described in Section 5.4.4. The planner model is responsible for planning the tasks required to follow the life cycles in the agent's agenda. Finally, the environment model is capable of searching (by consulting the agent platform's DF) for Operators and store them in a repository of known Operators. Information related to report-to relations with Reconfigurator and user agent are also stored in the environment model.

<table>
<thead>
<tr>
<th>model</th>
<th>functions</th>
<th>domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>communication</td>
<td>process MAP, report to reconfigurator instruct Operators, process reports</td>
<td>available message content ontologies</td>
</tr>
<tr>
<td>competence</td>
<td>MAP construction, operator negotiation, operate MAS, negotiate with reconfigurator</td>
<td>application configuration, current MAP coordination mechanisms, result set</td>
</tr>
<tr>
<td>self</td>
<td>life cycle management, instruct planner</td>
<td>&quot;Manager role&quot;</td>
</tr>
<tr>
<td>planner</td>
<td>task selection, plan tasks</td>
<td>agent's agenda</td>
</tr>
<tr>
<td>environment</td>
<td>search for operators consult DF</td>
<td>repository known operators, report-to relation with reconfigurator and user agent</td>
</tr>
</tbody>
</table>

Table 5.1
The five models of the Manager agent (according to the 5C Model, see Section 4.2 (p.66)) representing its capabilities split up in function and domain.

5.4 Levels of Interoperability

In order to have the agents "smoothly" collaborate with each other, we discuss the problem of enabling Interoperability. Interoperability includes how the agents can communicate with each other, when they communicate and what message content they use. In order to study this problem, we apply four levels of interoperability: technical, syntactic, semantic and coordination. The first three levels correspond to traditional interoperability structures in agent communication such as [Haustein and Luedecke, 2000, Bellifemine et al., 2001, FIPA, 2002a]. In these structures the emphasis is on message transport, languages and ontologies. We added the coordination level in order to regulate communication patterns and flow of information. By this, we made a framework to abstract technique, representation, concepts and strategy to enable interoperability.

Within the framework, we made the assumption that interaction between agents is based on message passing, meaning that the agents are not capable of, for example, invoking methods at other agents but have to explicitly state a question in a message (cf. [Wooldridge, 2002]). In order to enable agents to exchange messages, they need to agree on using the shared network protocols and message transport mechanisms. These decisions can be handled in the Technical interoperability (or transport) level. For example, all or a selection of agents can agree to use HTTP as information exchange protocol and TCP/IP as information (message) transport mechanism. Decisions related to envelope-encoding and message content languages are covered in the syntactic

\(^6\)MAS stands for Multi-Agent System
Interoperability level, such as agents using XML and FIPA-ACL. Semantic interoperability means that agents use shared ontologies such as domain ontologies [Fensel et al., 1999b]. Finally, coordination interoperability implies agents using the shared procedures, such as “every service has to register”, and sharing the notion of organizational roles, such as Manager and Librarian (cf. Section 3.4). The interoperability levels are summarized in Figure 5.2.

Although some topics seem trivial, they are briefly mentioned to indicate the required steps to enable interoperability. We describe the technology and methods involved in enabling interoperation in multi-agent systems, to show the complexity of having agents interact with each other. However, the point is, when using standards, agent designers only have to deal with the coordination level, which should realize smooth collaboration between the agents.

### 5.4.1 Technical Interoperability

Although it is possible to have a multi-agent system running on one machine, most multi-agent systems will be distributed over a network of machines. These agents have to exchange messages with each other, which involves sending and receiving parties. For that, an addressing mechanism and a message transport mechanism are required. In the IBROW architecture, we applied the FIPA agent communication standard [FIPA, 2002b]. FIPA uses HTTP over TCP/IP for message transport, meaning that agents use the standard Internet protocol, available on any web-enabled machine. Using “GET” and “POST” commands, “MIME encoded” information can be transported. The addressing mechanism is based on standard URL and IP addresses. For example, the Manager can be addressed using Manager@gaper.swi.psy.uva.nl. For more details, we refer to the FIPA specifications [FIPA, 2002b].

The use of standards is important, because standards decrease the amount of new technology that has to be introduced to agents and agent wrapper builders. Furthermore, standards such as HTTP, TCP/IP and URLs are well tested and have matured to reliable and robust means for information and data exchange. A survey on detailed technical agent communication issues can be found in [Huhns and Stephens, 1999, Labrou et al., 1999, Bellifemine et al., 2003].

### 5.4.2 Syntactic Interoperability

When having enabled message transport between agents within a network (i.e. message sending), we look how agents can compose and parse agent messages. First, the format of the message exchanged will have to be known by senders and receivers. Secondly, the agents need to have agreed on a standard format. Such a format is defined by FIPA, which provides a vocabulary for message formats. This vocabulary is called FIPA-ACL (Agent Communication Language).
which is loosely based on KIF [Huhns and Stephens, 1999, FIPA, 2002b]. A message written in FIPA-ACL is composed of two levels: message content and message meta information.

The message meta-information contains the addressing, such as sender and (intended) receivers. Furthermore, it provides information on the actual content of the message, such as what language was used for the content and what (message content) ontology is to be used to couple meaning to the terms used. Several content languages are allowed, such as RDF, XML and SL [FIPA, 2002b]

In the IBROW architecture, XML is used as content language, because the brokering service (i.e., a Prolog process) and the Reconfigurator use XML to reason about application configurations.

The actual content of messages can contain coordination information, such as instructions and reports, and information, such as input, support and output objects. These objects can be encoded and annotated using “MIME-types”. Instructions and reports are expressed in the XML language. An example of a message is given in Figure 5.17 (p.114).

5.4.3 Semantic Interoperability

In order to provide semantics to agent communication, ontologies can be used [van Aart et al., 2002a]. Ontology-based communication is discussed in detail in Section 6.3. Also the use of ontologies and agents is motivated by the development of DAML-Services7. The difference between our approach and that of DAML-S, is that we do not commit to a single representation language.

On the basis of ontology-based communication, a number of specialized ontologies are defined that are able to manage the diversity of information transportation within the IBROW system. An alternative is to define one central ontology that is able to cover all information flows. If all agents would have to commit to this single ontology, the agents would be equipped with knowledge they do not need to perform their activities. For example, an Operator would not need to reason about how to retrieve PSM descriptions from a PSM library. Furthermore, it is unlikely that agents that are built by different institutes will commit to a single ontology. This point is further stressed by Hendler, who has predicted that there will not be large centralistic ontologies, rather a lot of specialized ontologies [Hendler, 2001]. The use of a specialized ontology also adheres to the notion of separations of concerns. Light weighted and dedicated agents can be built, which can be easily maintained and even replaced if necessary.

In order to investigate what ontologies are required, we discuss an analysis of interactions between the agents within the IBROW architecture. The interactions are required to accomplish tasks within the IBROW system, such as “select PSMs from PSM libraries”. For this particular task an ontology is required that is able to express information related to PSM competences such as pre- and post-conditions. An interaction has an initiator (i.e. sender) and one or more responders (i.e. receivers). Within an interaction, messages are exchanged that use terms from ontologies. The result of the analysis is given in Table 5.3.

The ontologies are discussed in detail below. Some of these ontologies are based on the UPML framework. UPML provides a frame in which information related to competences and behavior of PSMs can be stored.

7 see, www.daml.org/services/
Interoperations within a Complex Multi-Agent Architecture

<table>
<thead>
<tr>
<th>Num.</th>
<th>Initiator</th>
<th>Responder</th>
<th>IBROW Task</th>
<th>Ontology</th>
<th>Exemplar terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>user agent</td>
<td>broker agent</td>
<td>delegate user goal</td>
<td>task</td>
<td>pre and post condition, input and output roles</td>
</tr>
<tr>
<td>2</td>
<td>broker agent</td>
<td>library agent</td>
<td>select PSMs</td>
<td>task-method</td>
<td>pre and postcondition, pragmatics</td>
</tr>
<tr>
<td>3</td>
<td>broker agent</td>
<td>manager agent</td>
<td>delegate configuration</td>
<td>process</td>
<td>primitive step, role, PSM</td>
</tr>
<tr>
<td>4</td>
<td>manager agent</td>
<td>operator agent</td>
<td>coordinate execution</td>
<td>operations</td>
<td>consume, produce,</td>
</tr>
<tr>
<td>5</td>
<td>operator agent</td>
<td>user agent</td>
<td>acquire input</td>
<td>domain</td>
<td>input</td>
</tr>
<tr>
<td>6</td>
<td>operator agent</td>
<td>operator agent</td>
<td>transfer intermediate objects</td>
<td>domain</td>
<td>intermediate objects</td>
</tr>
<tr>
<td>7</td>
<td>manager agent</td>
<td>reconfigurator</td>
<td>reconfigure application</td>
<td>process</td>
<td>step, role, PSM</td>
</tr>
<tr>
<td>8</td>
<td>manager agent</td>
<td>user agent</td>
<td>report results</td>
<td>domain</td>
<td>outcome</td>
</tr>
</tbody>
</table>

Table 5.3
Interactions between agents, associated tasks and ontologies. The relation numbers (① - ⑧) corresponds with the collaborations from Fig 5.2 (p.88).

**Task ontology**

The user agent is responsible for assisting the user in formulating goal specifications. The goal specification is delegated to the broker agent, using the task ontology. The interaction between the user agent and broker is represented by directed line ① in Figure 5.2. Within the interaction between the user agent and broker agent, the user agent uses concepts such as input and output roles and pre- and post-conditions to specify the goal of an application. The broker agent uses the goal specification as input to configure an application. For details on goal specification we refer to literature on UPML [Fensel et al., 1999b, Fensel et al., 1999a]. On how an application is configured, we refer to [Wielinga et al., 2003].

**Task-method ontology**

In order to configure an application, the broker agent selects PSMs from PSM libraries. The library agents provide the broker agent with PSM descriptions. The interaction is represented by the directed line ② in Figure 5.2.

The pre and post conditions concepts are required, to reason about the competences of individual PSMs. Pragmatics are required to reason about availability, performance and the configuration of PSMs. These competence descriptions are maintained by library agents, which can be queried by the broker using the task-method ontology.

The difference between the task ontology and the task-method ontology is that the task ontology describes the expected behavior of an overall system, the related domain and the data required to execute the system. The task-method ontology deals with domain independent competence descriptions, possible configurations and pragmatics of individual PSMs.

**Process ontology**

When the broker has completed an application configuration, it is delegated to the Manager, see the directed line ③ in Figure 5.2. An application configuration is expressed in terms of primitive steps, roles and PSMs. A sequence of primitive steps defines the application configuration. A primitive step represents the work to be done. The work means invoking a PSM using support roles and feeding input. Support roles are used to express PSM configuration, e.g. what knowledge base to use. The input and output exchanged between steps are expressed by intermediate roles. In fact, intermediate roles are used as transport objects between different PSMs.

The input of the application is expressed by input roles and the output by output roles. Every primitive step refers to the configuration of a single PSM. The first primitive step within
the application configuration, defines one or more required input roles. The output roles of the application are defined by the last primitive step in the primitive step sequence. The primitive steps between the first and last primitive steps use intermediate roles (i.e. transport objects) to exchange input and output roles. The terms used in an application configuration are covered by the *process ontology* (see Figure 5.3).

![Figure 5.3](image_url)

*Process Ontology, i.e. the application configuration schemas.*

**Operations ontology**  Based on the application configuration, the Manager constructs a MAP. Terms such as input roles and output roles are translated to input objects and output objects. These terms are captured in the operations ontology, which is based on the task-method ontology for coordination defined in Figure 3.3 (p.41). The involved Operators are instructed according to the MAP, see the directed line ④ in Figure 5.2 (p.88). After instruction, the Manager coordinates the application execution.

An instruction, exchanged between the Manager and Operators, contains the terms *consume_from* to expresses what *objects* i.e. roles to consume from which *Operator(s)*. The instructions are part of a coordination strategy as discussed in Section 3.2. When applying direct supervision, the Manager instructs the Operators to report (i.e. forwarding output objects) directly to the Manager. In standardization of work, the Operators are instructed to report to another Operator. The term *distribute_to* expresses what *objects* to distribute to which *Operator(s)*. These terms are covered by the *operations ontology*. After reception of the instructions, the Operators configure the PSM they represent.

When the Manager starts the actual execution, the Operators call the required inference functions. The outcome of the inference functions will be forwarded according to the Manager’s instructions.

**Domain ontology**  In order to execute the application, the Operators need initial information in terms of input objects from the user agent. The *domain ontology* describes the mapping between input objects and information related to the user domain. The information is stored
in the user's knowledge base (KB). The input objects are transported using agent interaction as denoted by the directed line ® in Figure 5.2.

When multiple Operators are involved, the Operators transfer intermediate objects to other Operators. These transactions are denoted by the directed line ® in Figure 5.2. When the execution is finished, the last Operator reports to the Manager as denoted by the directed line ® in Figure 5.2. From there, the Manager reports the outcome to the Reconfigurator (directed line ®). When the Reconfigurator is satisfied, the Manager will report the output to the user agent (directed line ®). A part of the trace of the interactions is illustrated in Figure 5.19.

The discussed ontologies are relative small in the sense that they contain a limited number of concepts and relations. The advantage is that the agents involved only have to be equipped with dedicated knowledge to reason about their domain. The price of this decision is that there is redundancy between the ontologies. This could cause additional effort for the maintenance of the ontologies.

### 5.4.4 Coordination Interoperability

Above we have analyzed the possible interactions between the agents and the means to enable these interactions. We will now discuss how the agents can work together in harmony.

The idea behind coordination interoperability is that agents have agreed to play organizational roles within a multi-agent system. Examples of roles are Manager, Operator and Broker. Accompanied with a role is the type of behavior. Among behaviors is reactive behavior, i.e. the agent will wait until another agent starts an interaction, and pro-active behavior, which means that the agent will take the initiative to start an interaction in order to fulfill its responsibilities.

We discuss the coordination between the agents in the execution space, i.e. the Manager and the Operators. Furthermore, the external behaviors of the Manager and Operators will be presented.

#### 5.4.4.1 Manager Operator Coordination

The role of the Manager is to coordinate the Operators, including telling Operators how to perform their work in detail, such as getting the input objects, how to transform input to output, and to whom to distribute the output. These instructions are part of the MAP that the Manager composes on the basis of the available Operators and the received application configuration. We will look at how the Manager and Operators collaborate, see Figure 5.4.

The idea is that the Manager first instructs the Operators with the use of an instruction. With an instruction, the Manager can implement a coordination mechanism. Next the Manager will request for actual invocation of PSMs. The two packages in Figure 5.4 show these two steps, i.e. instruction, and start operation.

The first step is instruction, which has as the intention to configure the PSMs and the data flows between Operators. This works as follows. The Manager will send a request to an Operator containing an instruction. Such an instruction contains the following four items.

1. From what agent to consume its input. For example, Operator B needs input from Operator A.
2. What PSM to invoke. For example Operator A will have to invoke a PSM which can be a Prolog function.
3. Configuration of the PSM, i.e. the coupling to a domain, which is in most of the cases a knowledge base.

4. To what agent to distribute its output. In case of centralized coordination, this will be the Manager. In case of decentralized coordination, it will be another Operator.

When the Operator has received the Manager’s instructions, it will try to configure its PSM. The Operator can respond with a report containing:

1. not_possible, which means that the Operator cannot get access to the requested PSM or the PSM is not available.

2. done, meaning the instructions are processed and the PSM is configured.

The next step of the Manager is to start the actual application execution, i.e. the MAP operation. For this the Manager will send a request to the Operator containing the job to be performed and the required input. The Operator can respond with:

1. failure\(^8\), meaning that the execution of the job failed, i.e. an exception raised by the PSM itself. For example, there could be something wrong with the received input.

2. result, containing the output of the PSM.

The collaboration described above is of a simple kind. More elaborate collaborations where the Manager and Operators go in negotiations are subject for further research. The aim of this collaboration is to show how the four interoperability levels fit on each other.

\(^8\)The difference between the “not_possible” and “failure” report is that “not_possible” means that it is not possible to invoke a PSM. The term “failure” indicates that it is possible to invoke a PSM, however that in execution time, the PSM has triggered an error message.
The next two sections will describe the individual behaviors of the Operator and Manager that implement this collaboration.

### 5.4.4.2 Operator Behavior

In Section 3.4.1, we introduced the behavior of an Operator, which consists of a composition of three life cycles, i.e. the platform life cycle, the application life cycle and the execution life cycle. For the Operators in the IBROW architecture, we adjusted the three life cycles to: platform life cycle, the application life cycle and the PSM invocation life cycle. The PSM invocation life cycle is a specialization of the execution life cycle. The behavior of the Operator is illustrated by means of a pseudo state diagram in Figure 5.5.

![Figure 5.5](image)

**Figure 5.5**

Pseudo state diagram showing states (rounded boxes) and transitions (arrowed lines) of the three life cycles of the Operator’s behavior within the IBROW architecture. This diagram is a specialization of the diagram in Fig. 3.15 (p.54).

The **platform life cycle** is the same as the platform life cycle described in Section 3.4.1.

The **application life cycle** starts when the Operator receives instructions from the Manager. The Operator will move to the configuration negotiation state. From this state, the Operator will try to configure the PSM, the Operator represents. If successful, the Operator will be part of a (larger) application and will wait until it can enter the **PSM invocation life cycle**. If the configuration fails, the negotiation will be aborted and the Operator will leave the application life cycle. Otherwise, the Manager will report to the Operator that the application execution is terminated. In this case the Operator will reset the configuration of the PSM and leave the application life cycle.

The **PSM invocation life cycle** will start, when the PSM is successfully configured and when the Operator has acquired input for the PSM invocation. An Operator can acquire input on a re-active and pro-active manner. These two modes depend on the instructions received by the Manager. Given the acquired input, the Operator will invoke the PSM it represents. The result of the PSM will be distributed according to the instructions of the Manager. After output distribution, the Operator will go back to the part of application state.

### 5.4.4.3 Manager Behavior

In Section 3.4.2 (p.55), we introduced the behavior of a Manager, which consists of three life cycles, i.e. the platform life cycle, the configuration life cycle and the execution life cycle. For the Manager in the IBROW architecture, we adjusted the last two life cycles to: MAP configuration life cycle and MAP execution life cycle. The pseudo state diagram for the behavior of the Manager is illustrated in Figure 5.6.
The MAP configuration life cycle starts when the Manager receives an application configuration from the Broker. The Manager will recruit Operators by consulting the agent platform’s AMS and DF to search for relevant Operators. When a set of candidate Operators is found, the Manager will start negotiations with the Operators. In case of success, the Manager will enter the MAP execution life cycle. Otherwise, the Manager will report to the Reconfigurator. From the wait for Reconfigurator response, the Manager can leave the MAP configuration life cycle or can receive a new configuration.

In the MAP execution life cycle the Manager will start from the configured MAS state of the MAP configuration life cycle and will start MAS operation by sending job requests to involved Operators. When the MAS operation has been finished, the Manager will report the results of the multi-agent system (MAS) to the Reconfigurator.

When designing a 5C agent, using the internal and external behavior, the self-model would contain the role and the goals of the agent. The planner model would contain re-active and pro-active behavior in order to follow the life cycles. Interaction between the agents, including the technical, syntactic and semantic interoperation, could be handled in the communication model. The environment model would contain models of the roles of other agents.

5.5 Implementation

In this section, we discuss how the agents within the execution space are implemented. Given the conceptual description of the agent components and behaviors we discuss how pieces of the agent architecture can be implemented. With this implementation, we have performed a number of experiments, which will be discussed in Section 5.6.

The main challenge is to implement agents that can operate in a distributed environment. However, this is not the only challenge we have to face. In fact, we also need to address the problems related to agents running on several machines that are distributed within a (possible) large scale network. Another challenge is set by the need to test and debug the agents, separately and in combination. For example, there is no possibility of a desktop GUI for every individual agent. In the remainder, we first discuss the applied technology set, i.e. the tools and technology that are used as the basis for the IBROW system. Next, we present the implementation of the basic IBROW agent, which is the parent agent of the Manager and Operators. After that, we discuss the
agent log, which is used to store communication traces and agent activity log. Finally, we present inspection tools that can be used to post-mortem inspect the dynamics of the agents within the system.

5.5.1 Technology Set

Below, we describe the basis of the IBROW implementation and we discuss the technological solutions we have applied, in order to address two important problems: the development of agents and the development of PSMs.

5.5.1.1 Agent Development

An important enabling factor for the development of multi-agent systems is constituted by the existence of a number of agent-oriented toolkits\(^9\) that natively provide basic services such as communication, life cycle management, yellow pages and so on.

In the IBROW system, we applied a popular agent toolkit: JADE (Java Agent DEvelopment framework) [Bellifemine et al., 2001]. JADE is a software framework that simplifies the implementation of multi-agent systems through middleware that complies with the FIPA specifications, a library of classes that developers can use or extend while creating agents and a set of tools that support the debugging and deployment phases. JADE agents communicate by exchanging messages in compliance with the FIPA ACL language. Furthermore, JADE supports the AMS (Agent Message Service) and the DF (Directory Facilitator), which represent the white and yellow page for agent (service) discovery.

Given the already existing basic agent libraries, we have built the IBROW agents on top of the JADE toolkit. The agents were developed in Java (JDK 1.3) and deployed as Linux services. This means that the agents can be remotely started, suspended and stopped.

5.5.1.2 PSMs

Several PSMs are available as Java libraries (i.e. Java packages). Examples are data transfer, database access, parsers and composers and content grabbing. PSMs for data transfer include FTP, HTTP, EMAIL clients and repositories. Amongst PSMs for Database access are standard JDBC (i.e. Java version of ODBC) couplings for several commercial and open source database implementations. Several packages are available for parsing and composing, such as javacc and Document Object Model (DOM)\(^10\). With these packages, parsers for RDF and XML can be applied. In order to fetch information from web pages, WebL can be used\(^11\). WebL is a scripting language that enables Java programs to extract information from web pages. For example, a WebL script to extract information from the search engine Google is given in Figure 5.7. In fact, WebL is an example of a wrapping technique and is more useful than the interface that Google offers, because it can easily be altered to query other web services.

The problem is that many PSMs are distributed and heterogeneous in terms of input, output roles and behaviors. In order to have an Operator representing one of more available PSMs, the Operator should have access to it. A solution is using PSM wrappers and PSM transducer (cf. [Genesereth and Ketchpel, 1994]), which can mediate between a PSM and other agents. The

---

9www.agentlink.org/resources/agent-software.html

10see www.sun.com/products/JavaCC and http://xml.apache.org/xerces2-j

11http://research.compaq.com/SRC/WebL/
transducer is capable of accepting messages from agents, translating them into the PSM interaction protocol and consulting the PSM. After the response of the PSM, the PSM transducer translates the response into the agent communication language and sends the resulting message to other agents. The advantage is that no knowledge of the PSM other than its interaction behavior is required and is therefore useful when the code for the PSM is unavailable to the Operator builder or too difficult to modify.

When the code (i.e. methods and calls) and state (i.e. data structures and knowledge bases) of a PSM are available, a PSM wrapper can be applied to directly examine and manipulate the PSM. Wrappers are more efficient than transducers, because there is less serial communication. Both methods are supported within the IBROW system.

An example of wrappers and transducers for a specific domain is given in Figure 5.18 (p.115).

Several off-the-shelf components are used as the basis for the implementation of the IBROW system. The most important reason is that when pursuing web and agent standards, components that already comply to these standards should be used.

5.5.2 Agent Implementation

All agents within the IBROW system are extensions of the IBROW agent. Within the IBROW system, there are agents developed in Prolog, mainly as transducer of PSMs written in Prolog, and agents developed in Java. In the remainder we focus on the Java agents, which are extensions of the JADE agent class. The JADE agent offers basic message handling, such as message receiving and sending. Furthermore, it offers a basic planning mechanism allowing the scheduling of agent behaviors. The IBROW agents offer services that are related to interoperability, such as life cycle management and message content ontology handling.

In the next three sections, we overview the internals of the IBROW agent, the Operator and the Manager.
5.5.2.1 IBROW Agent

Message transport is concerned with sending and receiving messages. Sending messages involves the construction of ACL messages as Java objects. ACL messages are represented by the Jade class, `jade.lang.acl.AclMessage`. An example is given in Figure 5.8.

```java
ACLMessage msg = new ACLMessage(REQUEST);
msg.setSender(this.getAID());
msg.addReceiver(new AID("receiver", "foreign-platform"));
msg.setLanguage("FIPA-SL");
msg.setOntology("Operations-Ontology");
msg.setEncoding("String");
msg.setProtocol("fipa-request");
msg.setContent(contentObject.encode("FIPA-SL", "Operations-Ontology"));
agent.send(msg);
```

Figure 5.8
Simplified pseudo code for configuration of an ACLMessage object.

A message is not sent directly to an agent, rather to the ACC (agent communication channel), which is to be seen as a message transport bus [FIPA, 2002a]. The ACC first tries to read the ACL part of the message. If the message is not correct, the sending agent receives a Failure message back. Otherwise, the ACC tries to deliver the message according to the receiver slot.

```java
public class Scheduler extends Thread {
    ...
    public void run() {
        while (state==running) {
            for every behavior {
                behavior.action();
            }
        }
    }
}
```

Figure 5.9
Basic behavior scheduling in simplified Java code.

Every agent makes use of behaviors. A behavior is a separate process that performs one or more tasks. Behaviors are implemented by behavior objects, which are used by the scheduler thread\(^{12}\). In Figure 5.9\(^{13}\), the basic scheduler process of an agent is given. As shown, every behavior is activated in a simple round robin scheduling method. Behaviors can be added and removed during runtime. In Figure 5.10 and Figure 5.11\(^{14}\) two examples of behaviors are given.

In the IBROW agent, we have implemented life cycle management as a composite state machine. The platform life cycle state machine is given in Figure 5.10. In every action, the behavior inspect the current state, and acts accordingly. An agent has a set of initial behaviors that take care

---

\(^{12}\) A thread is an active object with its own locus of control. Multiple threads can run at the same time.

\(^{13}\) This pseudo code is extracted from the original Java code. "{" represents "begin" and "}" represents "end".

\(^{14}\) In Java the else statement is not used in a conditional statement.
public class PlatformLifeCycle extends Behavior {
    private int state = 1;
    private boolean finished = false;

    public void action() {
        switch(state) {
            case INIT: agent.addBehaviour(new AMSRegistration());
            case AMSREGISTERED: agent.addBehaviour(new DFRegistration());
            case DFREGISTERED: agent.addBehaviour(new DFRegistration());
            case MODIFIED: agent.addBehaviour(new DFModify());
            case DESTROY: agent.addBehaviour(new DFDeregistration());
            case DFDEREGISTERED: agent.addBehaviour(new AMSDeregistration());
            case AMSDEREGISTERED: finished = true;
            ...
        }
    }

    public void update(int newState) {
        this.state = newState;
    }
}

Figure 5.10
State machine implementing the platform life cycle.

of AMS and DF registration. For examples of the AMS behavior, the DF behavior and example messages, we refer to [van Aart and Jansweijer, 2003, Bellifemine et al., 2003]. When a behavior wants to change its state, it uses the function update.

5.5.2.2 Operator

An Operator is implemented as a subclass of the IBROW agent, which means that its competences are implemented as behaviors. We discuss one of these behaviors in detail.

As described above, the Manager can choose to follow several coordination strategies. The behavior for an Operator for Standardization of Work is given in Figure 5.11. This behavior is composed of three functions, action, processProcedure and processInput. In the function action the Manager checks if a message is received. If so, the content of the message is inspected. If the message contains an instruction in the form procedure (according to Figure 3.3 (p.41)), the Operator has to adjust its configuration, using the function processProcedure. As shown, the "member" activity contains the name of the activity the Operator has to perform in the current configuration. Furthermore, the member distribute_to contains the address of the agent to whom the Operator has to send its output to. With this simple mechanism, the Manager can configure Operators to coordinate themselves following a sequential MAP.

If the message contains an object, the Operator has to perform an activity using the object, which is handled by the function processInput. In this function the Operator produces an output on the basis of the activity selected by the Manager. Next, the output is sent to the agent as specified in distribute_to.

15 A "member" in Java can be seen as an "attribute" of an object or a "slot" of a concept.
5.5.2.3 Manager

Given a broker configuration, the Manager configures a MAS plan. The implementation of this process is shown in Figure 5.12. the Manager starts a loop over the of primitive-steps in the received ApplicationConfiguration object. For every primitive step, the Manager first determines the type of service requested, in this case using the name of the service. Next the Manager looks for an Operator that offers the service requested in the primitive step, using the platform’s DF. In this case, we assume that the Operator will cooperate without negotiation with the Manager. In other cases the Manager will need to negotiate on attributes like price, delivery time and duration of the execution of the service. Finally, the Manager adds the service combined with the selected Operator to the Multi-Agent Plan (MAP).

Given the MAP, the Manager can choose between several coordination strategies, see also Section 3.3 (p.42). The generateMAP function will be part of a coordination strategy. As an example, we give the implementation of direct supervision in Figure 5.13. When the direct-supervision behavior is instantiated, the constructor init is called. In this function, the member of the behavior is configured using the generateMAP function, which fills the MAP object.

Next, the delegateNextJob function is called, where the “current” activity is selected, in this case the first one, combined with the selected Operator. Using the activity and Operator information a Job is constructed, which is sent to the Operator. From here, the Manager waits for an answer from the Operator. If it contains a report, the function ProcessReport will be called, that replaces the current-object with the Output object received from the Operator. Next,
public MAP generateMAP(ApplicationConfiguration appConf) {
    MAP map = new MAP();
    for every (primitive-step in appConf) {
        String service = primitive-step.getPSM().getName();
        Agent operator = DFService.lookup(service);
        map.add(service, operator);
    }
    return map;
}

Figure 5.12
Simplified Pseudo code for Multi-Agent Plan (MAP) generation on basis of an Application Configuration.

the processReport function is called. This process repeats until all activities from the MAP are delegated to Operators.

5.5.3 Agent Log

In order to be able to gather data about the behavior of the agents within the system, we implemented an agent log as a centralized database (or a blackboard) where remote agents (i.e. agents that are not located on the same machine) can store information. This information, such as communication traces and state transitions, can be used to study the dynamics of the system. The database is developed in MySQL\(^\text{16}\) and is hosted on an agentcities server in Amsterdam\(^\text{17}\). This dual Pentium-III server runs a Linux distribution.

The technique of logging is also applied to software services, such as web servers, that run on servers, which are located in remote locations. Using remote login techniques, log files can be inspected. For example, the web server package Apache\(^\text{18}\) makes use of an access log and an error log. The access log records all legal transactions regarding the web content. The error log records all failed transactions.

The IBROW agents log two types of information: information related to communication and information related to state transitions. Communication information consists of incoming communication (i.e. messages received) and outgoing communication (i.e. messages sent). The state transition information contains data on state transitions recorded by individual agents. Every log entry is accompanied by a timestamp. There are several techniques to fill an agent log, such as sniffing. In the sniffer approach, an external process monitors all communication and external behavior of agents. The problem with this approach is that not all events, such as state transitions can be externally monitored. Furthermore, agent engineers might not like to have an external process “spying” their agents.

Alternatively, (authorized) agent within the system access the agent log themselves. On the basis of remote connection techniques, the agents have access to the agent log database, where the agents themselves can choose what to record in the log. Others who want to inspect the dynamics of the multi-agent system can login to the database and perform queries on it. Examples of queries

\(^\text{16}\)www.mysql.org
\(^\text{17}\)http://gapers.swi.psy.uva.nl
\(^\text{18}\)www.apache.org
public class DirectSupervision extends Behavior {
    private MAP map;
    private Object current-object;
    private boolean finished = false;
    ...
    public void init(ApplicationConfiguration appConf, Object input) {
        this.map = generateMAP(appConf);
        this.current-object = input;
        delegateNextJob();
    }
    public void action() {
        ACLMessage msg = agent.receive();
        if (msg != null) {
            if (msg.contains(report)) processReport(msg.content);
            ...
        }
    }
    public void processReport(Report r) {
        this.current-object = r.getOutput();
        delegateNextJob();
    }
    public void delegateNextJob() {
        Activity a = map.nextActivity();
        if (a == null) { finished=true; return; }
        Agent operator = map.getOperator(a);
        Job job = new Job(a,operator);
        job.setInput(current-object);
        ACLMessage msg = new ACLMessage(REQUEST);
        msg.addReceiver(operator);
        msg.setContentType(job);
        agent.send(msg);
        ...
    }

Figure 5.13
Simplified pseudo code for Manager behavior when applying Direct Supervision.

are “is every sent message also recorded by the sending party”, “how many messages did the Manager send to Operators?” and “did every agent follow the life cycles?”. The log database contains two main tables, the message table and the state table. The message table is modeled after the format of ACLMessages. It contains fields such as sender, receivers, language, ontology, protocol and content. Other fields include the direction (i.e. incoming and outgoing) of the messages and timestamps (i.e. when the message was sent or received). For every message sent, a record is added by the sending agent and records are added by the receiving agents. By inspecting the table, communication traces can be followed. An example of a message trace is illustrated in Figure 5.14. The state table is modeled after a simplified model of a state machine. Every record in the table represents a state transition. For that, the table contains the following fields, begin-state, end-state, transition, timestamp, agentname. The transition contains the name of the event that triggered the shift from begin-state to end-state. Agent can log state transitions, so that their internal behavior can be monitored. The moment of transition
Figure 5.14
Screenshot of the Agent Log showing the results of a query performed on the agent log. The query is: "SELECT TIMESTAMP, DIRECTION, SENDERNAME, RECEIVERNAME, PERFORMATIVE, CONTENT from MESSAGE;".

can be stored in `timestamp`. The identification of the agent is stored in `agentname`.

The agent log is a database where information related to communication and state can be stored and retrieved. In the remainder we discuss a selection of tools that make use of the agent log. These tools are part of the `agent console`.

### 5.5.4 Inspection Tools

The agent console is a web service that offers a collection of inspection tools. There are two types of tools, textual and graphical tools. The textual tools present data retrieved from the agent log in the form of tables. For example, Figure 5.14 is generated from the tool that enables users to query the agent log, using own SQL statements.

The graphical tools make use of an external web service, `webdot`\(^\text{19}\). This web service can generate image files (i.e. Portable Network Graphics (PNG)\(^\text{20}\)) files that can be presented in a web browser. The agent console and webdot communicate with a simple graph language. Displayed objects, such as boxes and ellipses can be equipped with hyperlinks, meaning that if a user clicks an object, he is referred to a page with more details.

\(^{19}\)www.graphviz.org/webdot/
\(^{20}\)www.libpng.org


5.5.4.1 Communication Diagram

The communication diagram is a graphical representation of agents exchanging messages. Using the message table of the agent log, diagrams can be generated that show message interchange between a set of selected agents. Agents are expressed by ellipses and messages by boxes. Every message contains information on the sequence number of the message, the performative and a description of the content.

In Figure 5.15 a screen shot of a communication diagram is given. As shown, the user can specify a number of parameters that helps to narrow the size of a diagram. The boxes i.e. messages, are hyperlink sensitive, which means that they point to another web page. In this case the objects refer to another inspection tool, the message inspector. The message inspector shows (textually) all fields of a single record within the message table, in this case by the user selected.

Figure 5.15
Screenshot of a part of the communication between the the Manager and the Operators. The user can specify a number of parameters that helps to narrow the size of a diagram. The boxes, i.e. messages are hyperlink sensitive, where the hyperlinks refer to a page that contains containing detailed information related to a message.

Several observations can be made from inspecting communication diagrams. Deadlocks can be spotted, when a message is sent to an agent, but no outgoing messages follow. One expects the agent to reply to incoming messages to the sender or to other agents. If not, the communication trace ends at this agent, because the agent is down or it cannot do anything with the message. Otherwise, the sending agent has sent messages to the wrong agent, or with the wrong content.
One way to resolve dead ends within a communication trace is having agents always responding to messages, even when they are not the intended receiver of a message or understand the content (cf. [FIPA, 2002h]).

A life lock can be identified when two or more agents send failure messages stating that they cannot understand the message. It means that these agents send failure messages in a loop. These situations can be corrected by using the number-of-hops attribute of ACLMessages. This attribute counts the number of messages that are sent within a conversation. When this number exceeds a threshold, resolving actions can be taken by the agents involved. When an agent does not send or receive any message, this agent could be superfluous. In a multi-agent system design, several agents are identified, but in the execution of the system, agents can be unrequired. A reason can be that the competence of these agents is not required for the tasks and environment at execution time.

### 5.5.4.2 State Diagram

The state diagram is a graphical representation of the internal behavior of an agent. Simplified state machines can be drawn, based on the entries of the state table. The boxes represent states, the arrowed annotated lines represent state transitions. The annotation represents the identification of the transition. An example is given in Figure 5.16.

From a state diagram, parts of the internal dynamics of an agent can be studied. For example, a deadlock can be identified. A deadlock can be discovered by looking at the last state of a state diagram. If this state does not represent an end state, it means that the agent stopped processing at that point.

**Bottom Line** The discussed selection of tools has the purpose to assist agent engineers to inspect the dynamics of distributed systems. The graphical tools make use of webdot. However, webdot is a limited graphical engine. UML notation is limited too. Furthermore, the layout cannot entirely be controlled. The algorithm behind webdot tries to minimize the number of crossed lines. The use of link sensitive graphical objects (i.e. objects as hyperlinks) enables the user to "browse" traces within the agentlog.

The deployment of inspection tools as a web service can assist multiple (distributed) agent designers to inspect the dynamics of systems at the same time. Another advantage of a web service is that agent designers do not need to install special software in order to monitor the behavior of agents.

### 5.6 Classification of Conference Submissions

Document classification is an interesting domain for reuse of knowledge-intensive services and components. In this section, we will discuss a proof of concept that deals with classification of conference submissions. The focus of the discussion will be on the agents in the execution space.

A conference the size of ECAI2002 received over 600 submissions. The problem is that the program chair (PC) had to classify the submissions by hand in order to distribute them to reviewers. The idea is, to automate this process using a collection of configured PSMs. A submission for a conference consists of two parts:

**Submission form** This form contains the administrative details of the submitter (name, address, etc.) as well as information about the submission itself: title, abstract and keywords.
Submitted paper  The paper submitted as PDF. For scientific articles, it will contain a title, abstract, body and references.

The goal of the application is to determine for each submission to which “area” it belongs. An area is a sub-discipline within the field of the conference. For example, Machine Learning would be a sub-discipline of Artificial Intelligence and each Machine Learning paper should be distributed to the reviewer(s) responsible for this area. A sub-discipline may itself be further decomposed (indefinitely), e.g. Case-Based Reasoning is a sub-discipline of Machine Learning.

The PC (i.e. end user) has developed an ontology of the sub-disciplines it considers relevant for the conference and assigns keywords to these. The authors of papers enter a selection of these keywords in the submission form. Ideally, the keywords are sufficient to determine the correct area of a submission. In practice, authors often enter multiple keywords that are inconsistent (i.e. keywords that belong to more than one area) either because the content of the paper warrants it or to increase the probability of acceptance.

The strategy of the application is to look at multiple information sources to determine the area of a submission. Information sources are: title, keywords (as entered by the author), abstract and the paper itself. Obviously, reading the paper is more time-consuming than reading the abstract and if it is possible to derive the area from the keywords and/or abstract this is highly preferred.
The agents in the execution space start their operation using an application configuration. For that, we will briefly discuss the activities in the brokering space. Next we will show outcomes of the agents in the execution space.

5.6.1 Brokering

In order to configure the application, PSMs from three libraries were selected: the data transport library, the document analysis library and the classification library. After identification of these libraries, the broker retrieves PSM specifications. These PSM specifications can be matched with the goal specifications using different matching strategies, such as keyword match, simple task-PSM matching (on the basis of specification), and theorem proving based matching of competence. In our experiment, we restricted ourselves to the ECAI broker, which is described in [Wielinga et al., 2003].

The data transport library contains PSMs for P2P and web based file transport, i.e. download routines based on web and Operating System (OS) standards. The document analysis library provides PSMs that can perform various tasks related to documents (analysis, extracting features, reformulation, tokenizing, parsing, identifying phrases) 21. The PSMs have been designed in such a way that the output of one PSM can be used as the input to another component provided that the ontologies match. The PSMs can handle various representations, such as PDF, HTML, XML and plain text. The classification library (the IRS22) contains PSMs that can perform classification. These PSMs can be configured with a classification ontology, such as the AI classification taxonomy. Every PSM from the library is equipped with a PSM wrapper or PSM transducer, with which Operators can consult the PSMs.

<table>
<thead>
<tr>
<th>Step</th>
<th>PSM</th>
<th>Library</th>
<th>Features</th>
<th>Description</th>
</tr>
</thead>
</table>
| 1    | select-submission | data transport   | input=location  
output=document  
language=JAVA  
coupling=wrapping | select a submission from location      |
| 2    | extract fields     | document analysis| input=document  
output=fields  
language=Prolog  
coupling=transducing | select relevant fields and values     |
| 3    | extract-features    | document analysis| input=fields  
output=features 
language=Prolog  
coupling=transducing | construct features                 |
| 4    | classify          | classification   | input=features  
output=class  
language=Prolog  
coupling=transducing | classify features into a class       |

Table 5.4
Application Configuration for the ECAI application. This table is extracted from the original application configuration produced by the Broker, expressed in XML, in order to increase the readability. The bold values in the features column represent the initial input (in row 1) and the final output objects (in row 4).

The remainder of this work will describe how Operators are organized and managed in order to classify a bulk of scientific submissions for the ECAI 2002 conference in appropriate categories

21 www.swi.psy.uva.nl/usr/anjo/home.html
22 http://irs.kmi.open.ac.uk/
based on an AI classification taxonomy. We will not discuss how the broker has constructed a configuration, rather how the configuration is translated into a MAP, how this MAP is executed and the negotiation with the Reconfigurator. The configuration for the ECAI application as constructed by the broker is given in Table 5.4.

5.6.2 Execution

In order to illustrate the dynamics of the agents in the execution space, we have organized this section according to the states in the MAP configuration life cycle and MAP execution life cycle. The steps in the MAP configuration life cycle are MAP construction, Operator negotiation, configured MAS, MAS operation and wait for Reconfiguration response.

5.6.2.1 MAP Construction

Based on the application configuration, the Manager will construct a MAP, using the function as described in Figure 5.12. For every primitive step, the Manager will look for a suitable Operator, using the agent platform's DF, construct instructions and put these in the MAP. The selected Operators and instructions are captured in the MAP (see Table 5.5).

<table>
<thead>
<tr>
<th>Step</th>
<th>Operator</th>
<th>PSM</th>
<th>Operator Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>document-obtainer</td>
<td>select submission</td>
<td>activity=select-submission</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>input=uri</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>consume_from=useragent</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>output=XML document</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>distribute_to=field-extractor</td>
</tr>
<tr>
<td>2</td>
<td>field-extractor</td>
<td>extract fields</td>
<td>activity=extract-fields</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>input=XML document</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>consume_from=document-obtainer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>output=fields</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>distribute_to=feature-extractor</td>
</tr>
<tr>
<td>3</td>
<td>feature-extractor</td>
<td>extract features</td>
<td>activity=extract-features</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>input=fields</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>consume_from=field-extractor</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>output=features</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>distribute_to=classifier</td>
</tr>
<tr>
<td>4</td>
<td>classifier</td>
<td>classify</td>
<td>activity=classify</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>input=features</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>consume_from=feature-extractor</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>output=class</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>distribute_to=manager</td>
</tr>
</tbody>
</table>

Table 5.5
Multi-Agent Plan for the ECAI application. The original Multi-Agent Plan is expressed in XML.

In our experiment, we have chosen to let the Manager coordinate the multi-agent system using "standardization of work". Using this coordination strategy the Manager can delegate the control over the operation to the Operators (see also Section 3.3 (p.42)). Given the MAP, the Manager will start negotiations with the Operators.

5.6.2.2 Operator Negotiation

As described in Section 5.4.4.1, the Manager will start negotiations with Operators after construction of the MAP. Given the MAP, the Manager will inform the involved Operators on the basis of
instructions how to act. After reception of the instructions, the Operators will adjust their behavior according to these instructions.

The Operators use the instructions to configure their internal behavior, as described in Figure 5.11. The *distribute-to* term tells the Operator from which agent, what object to expect. For example, the agent *document-obtainer* will have to use the object *location* of the type *URL* as input for its PSM wrapper to invoke the PSM *select-submission*. The output (i.e. an XML document) of the PSM wrapper of the PSM will be sent to the agent as specified in the *distribute-to* slot. An example message is given in Figure 5.17.

![Example message](image)

**Figure 5.17** Example message, involving the Manager sending a message to document-obtainer containing an instruction. The instruction explains to the the document-obtainer that it should get its input (i.e. a URL referring to an PDF document) from the user-agent, that it should configure the PSM to the domain *DocumentClassification* and that it should distribute the output (i.e. a XML document containing the features of the PDF document) to the field-extractor.

### 5.6.2.3 Configured MAS

The Configured MAS is illustrated in Figure 5.18, which is arranged cf. the MAP in Figure 5.5. This diagram also includes the Operator negotiation, as described in the previous section. The lines of communication between Manager and the Operators as illustrated by numbered line ③ in Figure 5.2 resemble the lines of communication in the sequence diagram for Standardization of Work in Figure 3.11 (p.50). As shown, the type of interface between the agents and the PSMs is wrapper or transducer cf. Table 5.4.

### 5.6.2.4 MAS Execution

The next step of the Manager is to start the execution by sending the message "start operation" to the Operator: document-obtainer. The document-obtainer will ask (using a REQUEST message) the user agent to have the user to specify a location. The user agent responds (using an INFORM message) with the answer "URL". Document-obtainer will use its PSM wrapper to invoke the PSM *select-submission*, using the object *URL*. The response of the PSM wrapper is a collection of documents that contain submissions.²²

²²In order to illustrate the process, the list only contains one url.
The document-obtainer will send the documents one by one to the field extractor. The field extractor will activate its PSMs and send its results to the feature-extractor. The feature-extractor will forward its results to the classifier, which will on its turn forward its results to the Manager. The Manager will collect all results from the classifier.

The communication traces of the states Operator negotiation and MAP execution are illustrated in Figure 5.19.

When all documents are classified, the Manager will construct a report containing the application configuration and the result set.

5.6.2.5 Wait for Reconfiguration Response

Given the result set, the Manager will start a negotiation with the Reconfigurator. This negotiation is an instantiation of the FIPA ContractNet protocol. The Manager will PROPOSE the report (i.e. the broker configuration and result set) to the Reconfigurator.

The Reconfigurator can respond with an ACCEPT-PROPOSAL or REJECT-PROPOSAL. An ACCEPT-PROPOSAL means that the Reconfigurator is satisfied and that the Manager can report the results to the user. A REJECT-PROPOSAL means that the result set did not meet the criteria of the Reconfigurator and the Reconfigurator will instruct the Manager with an adjusted application configuration. Then the Manager will reconstruct a new MAP, and the execution process starts over again.
Communications Diagram from the Agent Log showing interaction traces between the agents within the execution space. The ellipses represent agents, the boxes represent messages and the arrowed lines represent the direction of the messages. The messages shown here are equipped with a sequence number, the speech act (or intention) and the actual content.

The results of the classification of the ECAI submissions are described in [Wielinga et al., 2003]. Several runs were required for the Reconfigurator to find an optimal configuration. We tried to describe the dynamics of the agents using the inspection tools of the agent console. As shown, the agents behave according to the “standardization of work” coordination strategy.

**5.7 Discussion**

In this work, we discussed a complex architecture in terms of service discovery, life cycles, coordination strategies and service invocation. The notion of separation of concerns was applied to focus on different aspects of the architecture. Every developed agent offers its own expertise such as PSM provider (i.e. library agent), invoker of PSM (i.e. Operator), coordinator (i.e. Manager), configurator (i.e. the static broker) and reconfigurator (i.e. the dynamic broker agent).

The intelligent agent metaphor enabled us to describe the services and their cooperation within the architecture as agents when represented by roles and behaviors. A lesson learned is that using separation of concerns instead of integration into one large monolithic system helped us to cluster heterogeneous services into one architecture. In order to enable the services (i.e. agents) to interact with each other, we applied common standards and available technology, such as FIPA compliant communication and procedures, agent toolkits and web technology.

We described the technology and methods involved in enabling interoperation to show the complexity of having agents interact with each other. When using standards, agent designers only have to deal with the coordination level of the interoperability framework. In the coordination level, designers only have to specify collaboration between agents. Therefore, they do not have
to fill in all the levels of the interoperability structures. However, the collaboration diagrams discussed in this work are of a simple kind. More elaborate collaborations where the agents can negotiate are a subject for further research. The design of ontologies for ontology-based communication is discussed in the next chapter.

There are several alternatives to our approach: one monolithic system, distributed objects and web services. Using a monolithic system approach all necessary components and services are integrated into one system. A number of the advantages are: there is a minimal need for interoperation and there is a central point of success and failure. The disadvantages are: all components need to be centrally gathered, which is not always possible, the application will become very complex, components written in different languages need to be translated or rewritten and only a small number of engineers can work on the system, due to integrity constraints. When using distributed objects, there is need for middleware, such as CORBA. The advantage is that objects can call methods of distributed object themselves, without explicit communication. The disadvantages are that standardization is achieved on a low level and there is a need for control processes that coordinate the distributed objects, because objects lack their own control. Finally, every component can be represented as a web service. The advantage is that SOAP is becoming a standard that is supported by many development kits. The reason for this is that SOAP is a light weight protocol that works with existing standards, such as XML and HTTP. The disadvantages are that web services lack interaction protocols, coordination mechanisms and the notion of (hierarchical) roles, such as Manager and Operator.

In the submission classification scenario we showed how an application configuration is translated into a MAP. The execution of this MAP showed how PSMs from different libraries can interact with each other. We had three libraries of PSMs to our disposal, i.e. data-transport, document analysis and classification. We did not investigate configuration and execution of services from other libraries, because the libraries were not available (i.e. only described on paper) or were not accessible (due to technical or semantic problems).

Using the agent console, we could inspect parts of the dynamics of the IBROW architecture. Despite the limited expressive power of the graphical inspection tools, we could gain insights into the communication and internal behavior of agents. However, the control over the agents is still a problem. When testing or debugging a multi-agent system, it is still difficult to start, restart, stop or suspend agents.

Although we showed some overall behavior (such as configuration and execution) of the architecture, most of the knowledge and behavior of the individual agents is hardwired. In order to have a flexible system, the agents should also be able to detect possible failures and find individual means (such as applying another strategy) or group means (consulting other agents) to overcome these failures.

In spite of the use of common standards (such as XML and FIPA) and technology we encountered most of the problems in the technical and syntactic interoperability layers. A possible improvement could be having less strict parsing technology. If a user makes a typing error, the user agent should be able to detect it and correct it. Furthermore, failing network connections, server failures and failing communication are inevitable in a distributed environment. Therefore, the agents should have error handling and fail-over mechanisms as found in grid computing. For example, if a PSM fails for some reason, an Operator should be capable to reschedule the execution to another Operator.

We foresee an agent-based market place consisting of provider agents, customer agents and broker agents. The broker agents deliver an intelligent service that enables third party knowledge-service reuse, where suppliers provide libraries of knowledge services adhering to some standard,
and customers can consult these libraries to configure a knowledge system suited to their needs by selection and adaptation. A customer in this context is a person/company who wants to solve a particular problem. Users who use the brokering service simply get results for their requests. It is not of interest for users how the brokering service acquired the result.