Opal: A Multi-Level Infrastructure for Agent-Oriented Software Development

M. Purvis, S. Cranefield, M. Nowostawski, and D. Carter
Information Science Department, University of Otago, Dunedin, New Zealand
Email: mpurvis@infoscience.otago.ac.nz

ABSTRACT

The Opal architecture for software development is described that supports the use of agent-oriented concepts at multiple levels of abstraction. At the lowest level are micro-agents, streamlined agents that can be used for conventional, system-level programming tasks. More sophisticated agents may be constructed by assembling combinations of micro-agents. The architecture consequently supports the systematic use of agent-based notions throughout the software development process. The paper describes (a) the implementation of micro-agents in Java, (b) how they have been used to fashion the Opal framework for the construction of more complex agents based on the Foundation for Intelligent Physical Agents (FIPA) specifications, and (c) the Opal Conversation Manager that facilitates the capability of agents to conduct complex conversations with other agents.

1. Introduction

The task of building complex distributed information systems is a major challenge facing the software engineering community, and an open networked environment, such as the Internet, offers additional challenges because of its dynamic and volatile nature. But whatever the particular circumstances of a given system, the design and management will be assisted if advantage can be taken of the following three notions [1]:

Abstraction: encapsulating information by defining modelling “chunks” that emphasise a few important details and suppress others.

Decomposition: hierarchical refinement by employing the notion of “divide and conquer”.

Organisation: the process of identifying and managing the interoperation of complex components.

Over the past fifteen years object-oriented and component-based techniques have been developed to take advantage of the first two of these notions. More recently, it has been suggested that the use of techniques associated with the concept of

software agents [2,3] are even more suited to the exploitation of the three notions listed above.

For such an argument to hold in practice, however, it is necessary that there be a suitable agent-building infrastructure available for software engineers so that they can employ agent constructs in the various ways that are envisioned. In particular to support *decomposition*, it is necessary to be able to use agents at various levels of modelling detail and refinement. In this way, a designer should be able to consider a system at any level of desired detail and think of that system in terms of agents (each of which could, in principle, be composed of smaller, internal agents). And the use of agent entities should not impose an unsatisfactory performance penalty on the designer who elects to use them. In addition, to take full advantage of the agent paradigm, support should be provided for the third of the notions listed above, agent *organisation.*

At the present time, however, there does not appear to be any agent-building toolkit or suitable infrastructural support that completely meets these basic demands and enables agent-based software engineering in the ideal manner envisioned. The main publicly available agent-building toolkits discussed in the literature [4-6] are focussed on the construction of systems whose individual agents have a relatively coarse degree of granularity and which are not intended to be refined into smaller agents. These systems may specify the use of interoperable and semantically rich string-based communication [7] or the support of high-level cognitive modelling [8]; and while valuable for certain situations, the use of such machinery for building smaller components, such as graphical user interface applications, may be irrelevant and impractical[1].

Existing agent-building toolkit systems are quite complex in their own right and have been primarily built using object-oriented software technology, rather than agent-based technology. Consequently they do not have the notion of agents built into their underlying machinery. The present paper discusses an approach for building complex software systems, whereby the concept of a modelling agent can be used at multiple levels of modelling and operational detail. With this approach, both fine-grained and coarse-grained agents can be employed where desired. In particular, one can use the approach to design an infrastructural agent toolkit that is to be used to support the general construction of agent-based systems. We discuss our work in this connection by describing the design and development of the Otago

---

1

MadKit [9] is a fine-grained agent-building toolkit that includes some ideas similar to those expressed in this paper, although to our understanding it has some architectural differences with our approach and has not been used as the basis for a coarse-grained agent-building infrastructure.

Agent Platform, Opal, an agent-building toolkit that has been designed using this multi-level agent-based approach and includes richer support for agent organisation by means of a novel agent conversation management module.

2. A Multi–level Agent Architecture

The overall goal of our approach is to use the notion of agency to model and build systems at any level of abstraction. This is achieved by instantiating the idea of an agent at the lowest level of operation so that it is practically realisable for efficient code execution but still retains enough of the features of "agenthood" that it can sill be considered to be an agent for modelling and design purposes. In order to facilitate the following discussion, we identify some terms to describe various aspects of our architecture:

- **Agent –** a persistent entity deployed on a multi-agent system. This can be considered to be an actor that plays one or more *roles* in a society of agents.

- **Micro-agent –** a particular type of *agent* that represents the lowest and most primitive level of agent instantiation.

- **Role –** a specification of a cohesive set of behaviours, functions, or services in the multi-agent society. Roles may be played by one or more agents in an agent system. Each agent playing a role may take a different approach to providing the role's services.

- **Responsive Agent** – an agent which does not control its own thread of execution, but simply reacts to the stimuli from the outside. Upon activation this agent can nevertheless perform deliberative computations, engage in social interactions, commit to or refuse to accept a particular goal given to it, or perform or refuse to perform a particular function assigned to it.

- **Autonomous Agent** – an agent which controls its own thread of execution. It actively pursues and maintains its goals, stimulates other agents, including responsive agents, and control and manipulate other agents (by playing the *Group* role).

- **Agent Group** – a role that provides an environment in which other agents (sub-agents) exist. This role is used for registration and discovery in a society of agents. It provides a mechanism for agents to locate each other based on the role they are playing, a role-based "yellow pages" service for micro-agents. Agents can register with more than one agent playing the group role.

- **Agent System** – any persistent society of agents. An agent system could have multiple groups, and specific groups or agents could be introduced, deployed or

redeployed, or killed at various times during the life of the system. An agent can be decomposed into smaller sub-agents that work together. When this happens, we can think of the original agent as having become an agent system.

A UML diagram of the key entities in the Micro Agent System is shown in Figure 1. There are two base elements in the Micro Agent System, *agents* and *roles*. Agents represent actors in the system that can play one or more roles. Roles are interface specifications of a cohesive set of services that may be provided by one or more agents, and each agent may take a different approach to providing the role's services in order to implement that role. An agent group is a role that provides an environment in which other agents (sub-agents) exist. Because an agent group is a role, some agents can contain other agents. This can be used as a hierarchical decomposition method for cases where it is logical to design agents in terms of a set of sub-agents. All agents belong to at least one agent group (an owner) that they live in; however top-level agent group does not have an owner, and this is effectively a recursion termination condition.

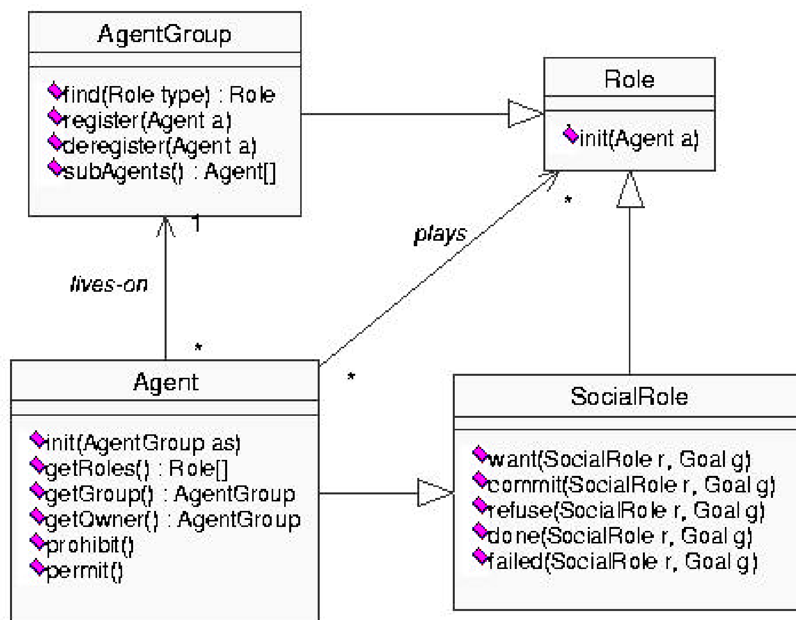

Figure 1. Micro-agent system design.

2.1 Micro-agents

Micro-agents exist at the lowest-level of agent-based abstraction in this architecture. In order to be efficience at this fine-grained level, they do not have all of the qualities often attributed to typical, more coarsely-grained agents. Those agents that exist at the higher levels of abstraction, such as those based on the Foundation for Intelligent Physical Agents (FIPA) [10] specifications, typically

engage in agent communication using a declarative representation for their messages that is based on speech-act theory [11]. Micro-agents, on the other hand, employ a simpler form of agent communication and, in addition, have more limited flexibility when compared to higher level agents.

Coarse-grained agents, such as FIPA-based agents, may make reference to ontologies (which characterise the terms and relationships mentioned in their messages), and they may reason about such ontologies or even adopt new ontologies for new agent conversations. Micro-agents, on the other hand, do not have ontologies in that sense but can be thought of as having an implicit system-level ontology that cannot be changed or reasoned about.

Micro-agents, being the closest entitiies to the machine platform, must be implemented on a specially design micro kernel. For example, for the case of the OPAL system, which is implemented in Java, the micro-agents are implemented by extending defined packaging and framework constraints, and they communicate via method calls. As a consequence, the micro-agents behave in strictly-defined and predictable ways and do not carry out runtime reasoning. Additionally, some micro-agents are responsive agents and do not own their own thread or threads of control.

Agents may be composed of any number of other agents or micro-agents. Non-primitive micro-agents are composed only of micro-agents. The same agent-based modelling approaches apply in the same way to both coarse-grained agents and to micro-agents – the same design methodology, role-oriented and society-oriented techniques apply equally to coarse-grained agents as to micro-agents. Agent-oriented decomposition and role-based modelling are independent of the deployment scale. All the roles can potentially be played by micro-agents or coarser-grained agents with a similar result. However the key advantage of micro-agents is that for small-scale systems they will radically out-perform coarse-grained agents at runtime.

2.2 Communication

For traditional communication in multi-agent systems, peer-to-peer asynchronous message passing with a formal agent communication language is used [7]. A message is embedded inside an envelope, which contains routing information, identification of the receiver and the sender, and the content of the message. The message content, which can be expressed in one of a number of possible content languages, makes references to terms formally defined in an ontology, and can be sent in the context of an ongoing interaction protocol [10] or conversation. A conversation in this context is a sequence of message exchanges which may span multiple interaction protocols and multiple agents.

In order to maintain the spirit of inter-agent semantic communication as expressed in speech-act theory, micro-agents have been designed so that they

communicate using messages of the following types: directives, assertives, expressives, commissives, permissives and prohibitives [12]. This does not represent a true implementation of natural language communication, but instead uses names derived from the discipline. The intention is to enable the developer to employ a mental model of language-based agent communication when micro-agents are used.

The social aspects of the agent interactions are captured in the SocialRole (Figure 1), which contains all the primitive type of communicative acts discussed above. The communicative acts (performatives) are implemented as simple method calls with a special argument list. The performative is represented as the method name itself, while the sender is identified by the first parameter, and the message content is represented by the goal argument. Goals together with Roles can be designed in UML, which then maps directly to the implementation via classes and interfaces.

2.3 Implementation

The current implementation of the micro-agents and the kernel which supports them is written in the Java programming language. This elevates existing object-oriented design patterns up to a useful agent-oriented abstraction level. However the Java programming language imposes some constraints of what can be done, and consequently some of the patterns that are desirable from the agent-based perspective cannot be implemented in a straightforward and efficient manner. Some of the most notable problems are:

An agent dynamically playing different roles maps to a class implementation that can play several differing interfaces at runtime. This is not possible without the inefficient dynamic proxy mechanism introduced in Java 1.3.

An agent will often need to identify the sender of a message, but using Java it is impossible to identify the caller of a method without using an additional formal argument in the method.

Java reflection, needed to discover runtime agent capabilities, is inefficient.

Despite these constraints, we believe that there are good reasons to implement an agent based system in the Java language, and we also believe that we have successfully implemented most of the features which lie at the heart of agent-oriented software engineering. The intention has been to provide a micro-agent and kernel framework which is as efficient as possible so that the micro-agent message operation involves little more overhead than a normal virtual method call in Java.

2.4 An illustrative example

To demonstrate the use of micro-agents from the Java developer's perspective we have developed a simple "Hello World" example. The Hello World agent system consists of four agents, the User agent, HelloPrinter, Dumper, and DataCollector, which are organized into two groups, as shown in Figure 2. The User agent is a social agent which wants the Hello World data to be printed. HelloPrinter is a social agent which can achieve the goal of printing "Hello World". Dumper is a responsive agent which can dump data to a system output port, and DataCollector is a simple responsive agent which can provide data, and in this scenario it provides the static "Hello World" string.

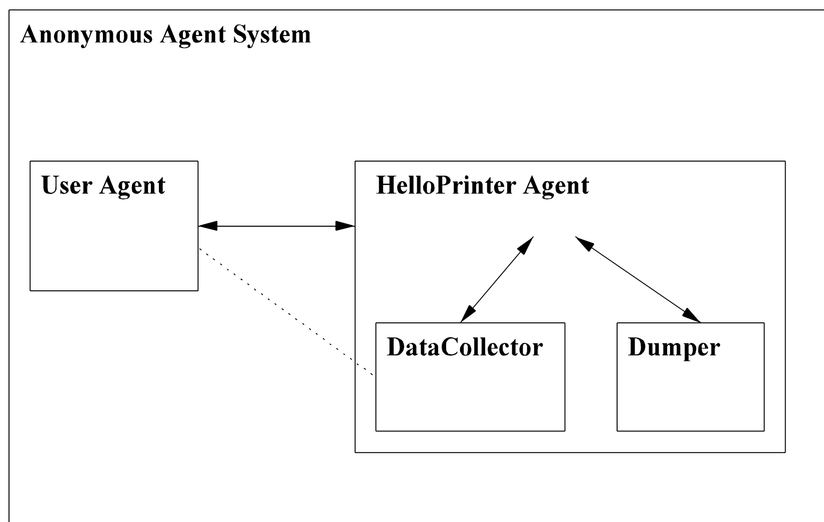

Figure 2. 'Hello World' example.

By separating these four different aspects of the system, different agents implementing the same roles can be plugged in without affecting the rest of the system. This allows us to have the following collection of agents,

Dumper agents, which dump to the screen, to a file, or to some GUI-based output.

Several DataCollector agents, one collecting data from the user sitting in front of the console, others from a file, telephone or GUI applications.

These extensions could be added during runtime by plugging in different agents dynamically, without affecting the rest of the system.

Setting up the Hello World system in Java could look like this:

```
public static void main(String[] args) {
  // the process of loading and creating new agents
  // performs registration and initialisation
  Agent hello = SystemAgentLoader.loadAgent( new
                                HelloPrinterImpl());
  Agent user =  SystemAgentLoader.loadAgent( new
                                UserAgent());
  // add subagents to hello agent
  Group group = hello.getGroup();
  AgentLoader loader = group.getAgentLoader();
  loader.loadAgent( new DumperAgent());
  loader.loadAgent( new DataCollectingAgent());

  // go!
  Goal g = HelloPrintedGoal.instance();
  hello.want(user, g);
}

// Hello Printer Role Implementation
public class HelloPrinterImpl
  extends DefaultSocialRoleImpl
  implements HelloPrinter {

  public void want(Agent a, HelloPrintedGoal g){
    // machinery to achieve goal
 }
}
```

The code above shows an example of a top level main method which sets up agents in different groups. One group is the top level agent group containing the User agent and HelloPrinter agent. The second group is contained within the HelloPrinter agent – it controls and manipulates a DataCollector agent and a Dumper agent.

Table 1. Performance for 10,000 'Hello World' iterations.

Implementation	**Time (ms.)**
Java	550
Opal Micro-agents	600
MadKit	15,000
JADE	122,000

To demonstrate the merit of the micro agent approach four different Hello World implementations have been developed and timed. The first implementation was a simple call to the Java method {`System.out.println("Hello World");` and the second the micro-agent implementation as described above. The third was an implementation using the the same agent-decomposition as the micro-agent example, but with the MadKit [9] agent toolkit. The fourth was a similar implementation using the JADE [4] agent toolkit. The results of timing 10,000 Hello World requests are shown in Table 1.

The JADE agents actually did not perform any message processing or parsing during the tests, and there was no data conversion performed for a given transport. It ran on single virtual machine and all message passing was done via a simple Java RMI mechanisms. This shows that coarse grained agents may in some cases be 200 times slower than our micro-agent implementation and confirms that FIPA-like agents are not likely to be suitable for fined-grained, simple and efficient system components.

2.4 Micro-agent Applications

In this subsection we discuss how the micro-agents can be used to build applications using an agent-oriented approach. In Section 3 this discussion is carried further by describing the development of the Opal system using micro-agents.

An initial agent-oriented model of a robot is shown in Figure 3. This model identifies the role of an robot, interacting with an environment and receiving instructions from a human operator. It would be possible to go from this model directly to an implementation of the robot as a coarse-grained agent. This implementation would be monolithic, and it is likely that further design, possibly using object-oriented methodologies, would be required to provide a decomposition from this high-level model to implementation level components.

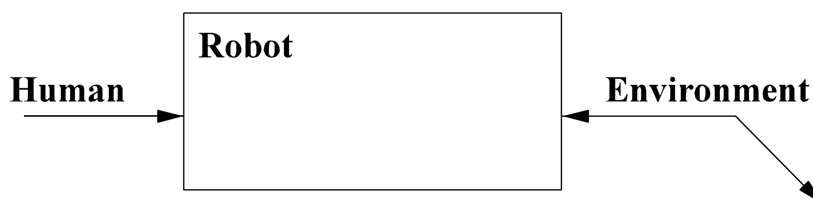

Figure 3. An initial agent-oriented model of a robot.

Figure 4 shows a more detailed decomposition of the robot. Several autonomous, concurrently running and communicating sub-roles are identified, a sensory processor, task scheduler, and meta-level processor. Three independent effectors are also identified, the arm and the left and right wheels. These components

may themselves be further decomposed, the figure shows only the decomposition of the task scheduler. Three reactive components of the task scheduler are identified: a command analyser, action planner and action executor. An example scenario could be the operator asking the robot to move to a particular location. This directive would be processed by the command analyzer, then the planner would create a plan to reach the location, and then the executor would send instructions to the wheels and arm to move to the location.

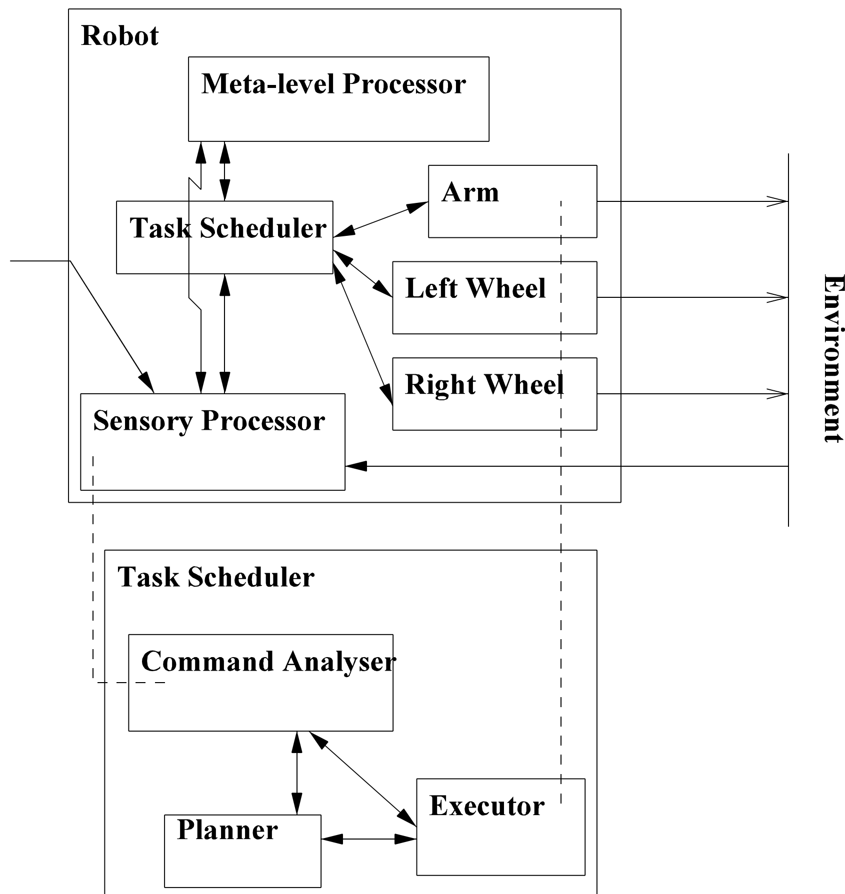

Figure 4. A refined agent-oriented model of a robot.

The level of detail expressed in this model would not be practical to implement using traditional coarse-grained agents – as the Hello World example has shown this would be too inefficent. Although the initial model for the coarse-grained implementation was agent-oriented, further refinement of such a model using traditional agent development technologies would likely require alternative design methods to be used. On the other hand, using the micro-agent approach described in this paper, a more-detailed level of design can be achieved with agent modelling employed all the way down to the implementation level.

3. Opal: the Otago Agent Platform

As discussed in Section 2.1, micro-agents are presented without much of the coarse-grained machinery often associated with "intelligent" agents. While micro-agents have been found appropriate for closed systems, such as those discussed in the previous section, many interesting agent-related research areas involve open systems, where the agents are typically coarse-grained, heterogeneous entities that make use of different languages and ontologies.

The Foundation for Intelligent Physical Agents (FIPA) is developing open specifications for such coarse-grained agent systems [10]. A key part of this work has been the FIPA Abstract Agent Architecture (FAA), which is an abstract specification of the infrastructure necessary to provide a suitable platform on which such agents can exist.

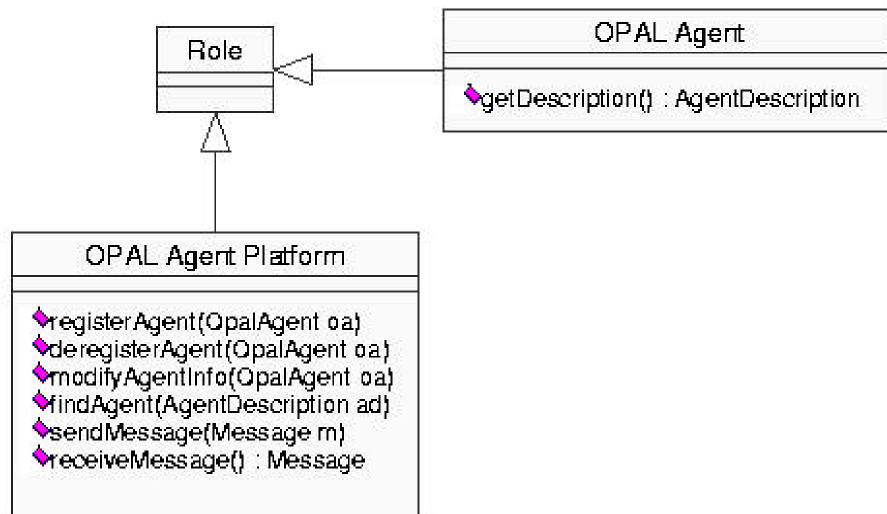

Figure 5. Micro-agent roles used by the Opal system.

This section discusses the Otago Agent Platform (Opal) which provides a concrete instantiation of the FAA, as well as other tools and utilities useful for the development of agent-based systems. Opal is a system which is based on and uses the underlying micro-agent kernel implementation. There is considerable amount of infrastructure specified by FIPA for coarse grained agents that does not exist in the micro-agent system as we have described it. To provide this additional capabilities for FIPA-type agents, specialised micro-agents need to be introduced. The Opal system is therefore designed to be a combination of these specialised micro-agents, that together provide for FIPA functionality.

3.1 Opal Architecture

An important concept of the FAA is the idea of an Agent Platform (AP) – this provides environmental support and the basic services for the agents deployed on it and it also provides a directory service to agents outside the AP. The Opal AP is implemented as a micro-agent playing the Agent Platform Role (see Figure 5). The key services that the AP provides are inter-platform message transport via the Message Transport System (MTS), agent management and a white-pages directory via the Agent Management System (AMS), and yellow-pages directory services via the Directory Facilitator (DF). These three logical capability sets are implemented in the Opal AP as separate micro-agents.

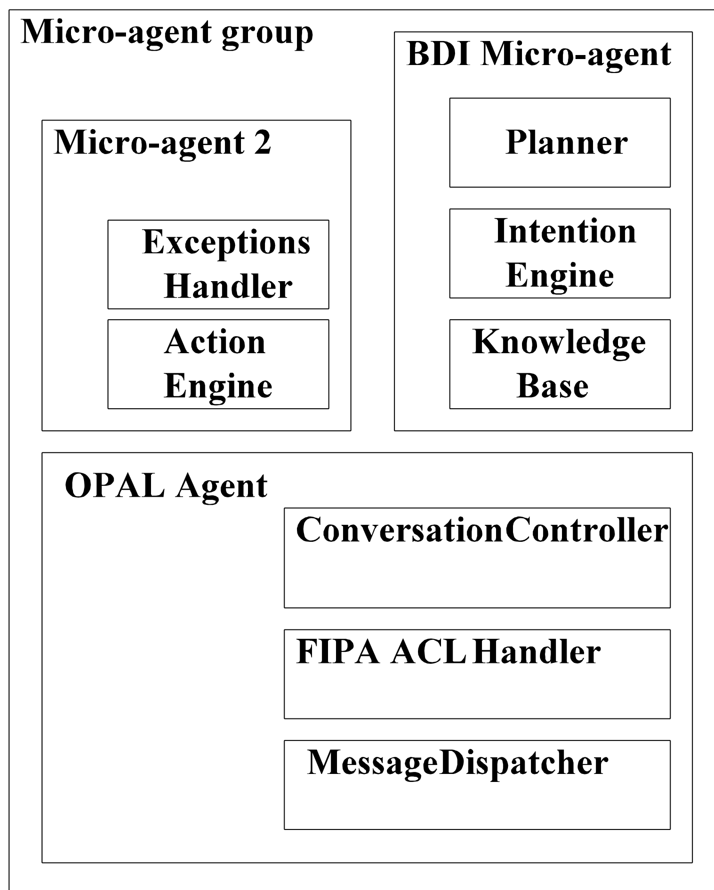

Figure 6. The Opal FIPA platform and Opal agents.

The AP Role implementation does not itself perform the bulk of the processing required for the actions of the AP Role (see Figure 5), rather its tasks are delegated to the three contained micro-agents. To register an agent, the MTS needs to know about the agent so it can receive messages for it, the AMS needs to add the agent to its white-pages directory and the DF needs to add the agent to its yellow-pages directory.

It is convenient for developers to specify the receiver of a message using a simple name. For the platform to send the message the transport-level address, which might be a CORBA IOR, Java RMI address or even an email address, it must first be found using the AMS, and then the MTS is used to send the message. When performing an action, the Agent Platform agent has the responsibility for ensuring that the correct sequence of sub-actions gets performed, but does not perform any of these sub-actions itself.

Aside from the micro-agent role representing the agent platform, an Opal system needs to contain the higher-level coarse-grained FIPA agents that exist on the Agent Platform. The micro-agent roles representing FIPA agents can contain a variety of sub-agent roles. An agent that contains no sub-agents is provided with only the ability to send and receive messages. Figure 6 shows a single FIPA agent that uses a conversation controller micro-agent role to keep track of conversations that the agent is involved in. The conversation controller requires that some micro-agent exists that is able to play the message dispatcher role. Another FIPA agent may require the Belief-Desire-Intention micro-agent role to enable it to be developed using the BDI model.

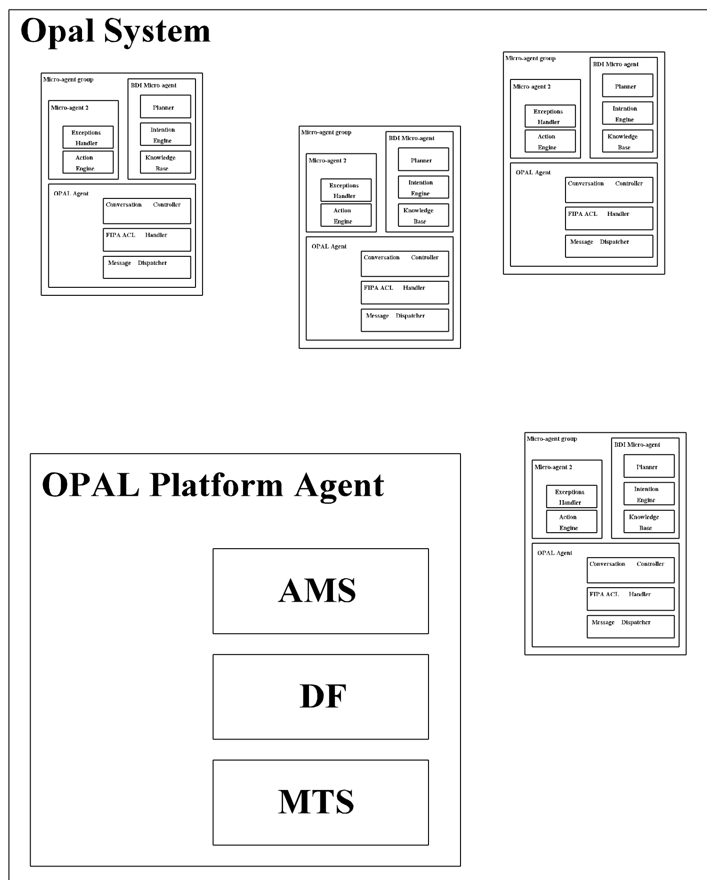

Figure 7. Example of an Opal system.

A complete Opal Agent System with associated FIPA-specified services is depicted in Figure 7. Individual Opal Agents of the type shown in Figure 7 can all access an Opal Platform Agent. The Opal Platform Agent contains individual micro-agents that implement the FIPA-specified services of the Message Transport System (MTS), the Directory Facilitator (DF), and the Agent Management System (AMS). The Opal Management Console, which facilitates user monitoring and control of the Directory Facilitator, agent messages, and the AMS, is shown in Figures 8 and 9.

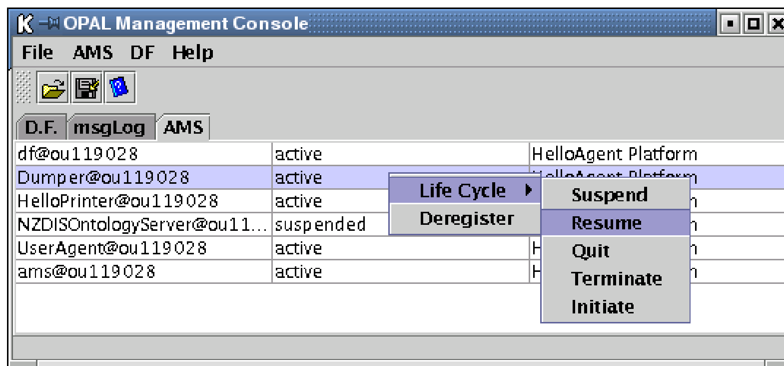

Figure 8. Opal Management Console: AMS.

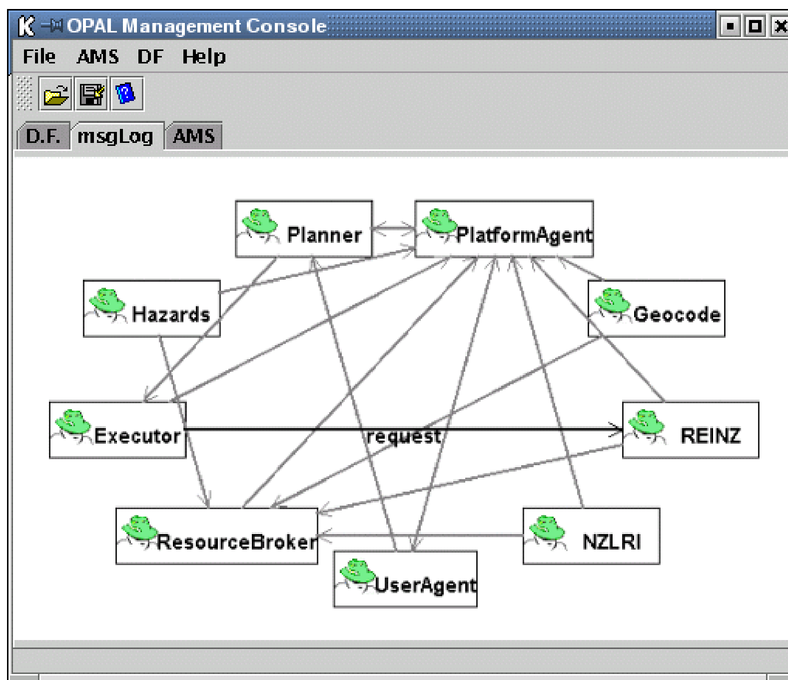

Figure 9. Opal Management Console: Message Log.

3.2 Message Transport and Dispatching

Transport services are the lowest level services provided usually by the Agent building toolkit or framework, to enable inter-agent communication and message passing. Most of the existing agent frameworks suffer, however, from different levels of incompatibilities, because they all provide their own customized implementation of message transport and dispatching. Opal differs from its predecessor agent frameworks in this respect, by employing the emerging industry standard for lower-level standard services, JAS.

JAS, which stands for Java Agent Services, is an effort to define an industry standard specification and API for the development of network agent and service architectures (for more details see http://www.java-agent.org)[2]. Opal employs a modular implementation approach to transport services not offered currently by any other agent framework. Transport services are pluggable, and thus new transport implementations can be seamlessly integrated into the platform when needed. By default, Opal provides implementations of the two main transport protocols used by FIPA platforms, FIPA JAS RMI-based and FIPA2000 IIOP-based transports, which can be plugged in and used in any JAS-compliant platform.

3.3 Interaction Manager

Opal implements several standard FIPA interaction protocols, and there is a special entity which handles dispatching and switching the state of a conversation based on the type and properties of received and sent messages. We refer to this module as the *interaction manager*, as distinguished from the *conversation manager* which is described in the following section.

The basic abstract role is an Interaction Tracker, which is specialized by concrete roles from all predefined FIPA interaction protocols, like FIPARequestTracker, FIPAQueryTracker, etc. Other agents can use the concrete instantiations of those roles, which are played by the dynamically created agents. The actual InteractionManager is responsible for starting, monitoring and controlling all those instantiations.

The system is flexible enough to cope with different aspects during runtime, for example a specialized Interaction Manager could be used to monitor and reschedule

---

[2] JAS does not have a formal specification yet, but there exists a preliminary proposed API set and reference implementation which Opal currently follows. Opal development efforts will continue to follow closely JAS efforts in order to maintain compatibility with it and ensure an up-to-date implementation.

priorities of the ongoing interactions, keep track of exceptions and delegate them to specialised units elsewhere in the system.

4. Opal Conversation Manager

A conversation is an ongoing sequences of messages which can span multiple agents and multiple interaction protocols. To manage the complexity of these conversations, a special infrastructural component must be modelled, implemented and deployed. There are many aspects which need to be addressed, including tracking individual interaction protocols and their violations, keeping track on the context and the subject, timing out late responses, allocating resources to more important tasks, etc. For the purposes of the following discussion, we identify some fundamental terms used when discussing agent interactions[3]:

communicative act – a special action type in the speech act theory that represents a basic building block of the dialogue between agents and has a well-defined semantics independent of the content of the action. There are currently two major specifications, the FIPA Agent Communication Language and the Knowledge Query and Manipulation Language (KQML) [14].

protocol (interaction or conversation protocol) – the template of the communicative acts sequence.

conversation – an instance of a conversation, a particular sequence of communicative acts.

policy (interaction or conversation policy) – strategies, guidelines and constraints guiding a conversation.

From our experience we have observed that agent conversation modelling can be decomposed into several separate layers. The first, most basic layer, is a *protocol layer*. A conversation protocol is a template of sequences of expected communicative acts organised into roles. This definition is compatible with the definition of a protocol specified by FIPA as: “a common pattern of dialogues used to perform some generally useful task; the protocol is used to facilitate a simplification of the computational machinery needed to support a given dialogue

---

3 Note that the terminology used by us does not necessarily match the terminology from other publications in the field, where protocol, policy, and conversation are very often used interchangeably. In particular the notion of conversation policy from [13] is equivalent to conversation protocol in our terminology.

task between agents; simply: a dialogue pattern" [10].

On top of that layer another layer is constructed: the *conversation* layer. A *conversation* is a particular instance of a protocol or set of protocols; it is an ongoing sequence of messages exchanged between two or more agents. This definition also complies with the one defined by FIPA: "an ongoing sequence of communicative acts exchanged between agents relating to some ongoing topic of discourse" [10]. However, we exclude from our usage the possibility of conversations being constructed from arbitrary chosen acts not conforming to a formal protocol.

The final, third layer is called the *policy* layer. This is the layer which would, with other approaches, be left to the agent application to coordinate and not be included explicitly in the conversation modelling process. However we feel that it is more appropriate to treat it as closely related to the conversation layer. A conversation policy is a collection of rules and interaction specifications that guide a particular path or trajectory in a conversation space. A policy defines the details concerning the conversation is handled by interested parties. Thus each *protocol* defines a space of possible sequences of communicative acts; each *conversation* follows one trajectory from this space, and a *policy* guides a particular conversation.

For example: imagine one protocol defining two roles, *buyer* and *seller*, and a sequence of acts for the buyer: *ask* (ask a seller for a particular goods delivery), then *accept* (accept the price and buy goods) or *reject* (reject goods, do not buy). And for a seller, the possible answers to the buyer *ask* action could be: *propose* (propose the goods price) then *sell* when accepted or do nothing when rejected. This simple specification is a protocol. Two participants have to follow a *protocol* to form a *conversation*. One possible *policy* (strategy) for the buyer would be to ask for goods from several other agents concurrently, and accept the lowest price and reject all the other proposals. That would dynamically create a relatively complex conversation involving several selling agents and a single buying one. A simpler strategy would be to *ask* only one selling agent, and *accept* or *reject* the proposal given by this agent, then start over a new conversation by issuing yet another *ask* to another potential seller for a proposal if the first iteration was finished without making a deal.

Policies may be implemented simply by set of rules, or, in more complex cases, they may have their own complex protocols that exist and change state in parallel with the immediate context of an ongoing conversation. Under these more complex circumstances, there might be a "policy-level interaction protocol" (another protocol, but at the policy level). It is under these conditions that we can benefit from having another modelling layer at the policy level, above that of the ordinary conversational modelling layer. The two layers can be joined together by representing them both as a coloured Petri Net (see Section 5).

Suppose, for example, we have a conventional conversation protocol involving

two players playing a game of chess in a chess tournament. There are possible rules for legal moves and legal responding moves by the opposing players, which would be described by this conversation protocol. But existing above that level of abstraction is another level of discourse that can take place during the game. Suppose one of the players has a question concerning the official rules of the game and wants to have a ruling made by one of the tournament judges. Or suppose one of the players at some point wants to take time out from the game and halt play so that he or she can drink water or attend to some personal needs. These kinds of 'interrupt' or 'exception' are common to many kinds of interactions and can take place at almost any time. The discourse involved in these interrupts are usually "off-topic" from the context of the immediate conversation, and in fact they are often *about* the conversation that is taking place (such as the chess player who may accuse his opponent of breaking the conversation protocol rules associated with playing the game of chess). Since they are likely to be "off-topic" and can occur at any moment, it can be tedious to include these kinds of conversational strands in the given (domain-specific) conversation protocol. To do so would "clutter" the visual simplicity of the original conversation protocol and would lessen the value in providing a easy-to-comprehend visual modelling representation of the interaction. On the other hand, to leave out the possibility of representing such events is to ignore the possibility of their occurrence and consequently means that there is a failure to model the world adequately so that its essentially contingent nature is recognised. Our solution is to model these kinds of interactions that can guide, interrupt, or redirect existing conversations by representing them as another, parallel modelling layer above that of the existing conversation layer. This idea was suggested in [15] for specific types of conversation, but we have generalised the notion and incorporated it into a Petri Net representation. Thus a *conversation* is a combination of *protocols* being instantiated and manipulated by a particular *policy*.

4.1 Conversation modelling

A number of modelling techniques have been employed to keep track of agent conversations, including Deterministic Finite Automata [13,16], Enhanced Dooley Graphs [17], and extended UML [18]. We use coloured Petri nets [19,20], because their formal properties facilitate the modelling of concurrent conversations and policies in an integrated fashion. Coloured Petri nets are similar to ordinary Petri nets in that they comprise a structure of places, transitions, and arcs connecting those two types of elements. In addition, coloured Petri nets have structured tokens and a set of net inscriptions (arc expressions, guards, and place initialisations) which can be evaluated to yield new net markings when transitions are fired. In the context of agent conversations, Petri net tokens represent messages, arcs represent message passing and delivery mechanisms, and transitions represent message processing units. Agent roles are organized into subnets, and roles are represented graphically by separating them with horizontal dashed lines. Arcs crossing role boundaries, i.e. arcs which cross dashed lines, represent physical message passing actions (the

process of sending and receiving a single message in the agent system). The arcs within roles are left up to the implementation and usually, for efficiency purposes, are implemented as method calls. Places represent message containers or intermediate containers, and usually do not map in the implementation to anything in particular, unless the Petri Net model is mapped directly to a Petri Net implementation (as in the case of Opal). Then a place is an abstraction of a message folder, containing processed or being-processed messages.

There is always one initiator of a conversation, a role which starts the conversation by issuing the very first message, and this role (and only this role) always has the *Start* place, which enables the very first transition to fire. All roles have separate dedicated *Terminated* places, which collect the tokens when no further message processing is scheduled to occur.

A conversation is a whole Petri Net composed of a set of subnets (i.e. protocols), where at least one role has the *Start* place (initiator) and is connected to an arbitrary number of other conversation participants. A conversation state is a current net marking. A conversation policy may, in straightforward cases, be encoded via arc inscriptions and guards inside roles of existing conversations. In more complicated cases, a conversation policy can be encoded as a parallel Petri Net that lies above the existing conversation protocol and represents exceptional, or "off-topic" conversational elements that may take place at various times during the ordinary conversation. See Figure 10 for an example of such a policy-level Petri Net.

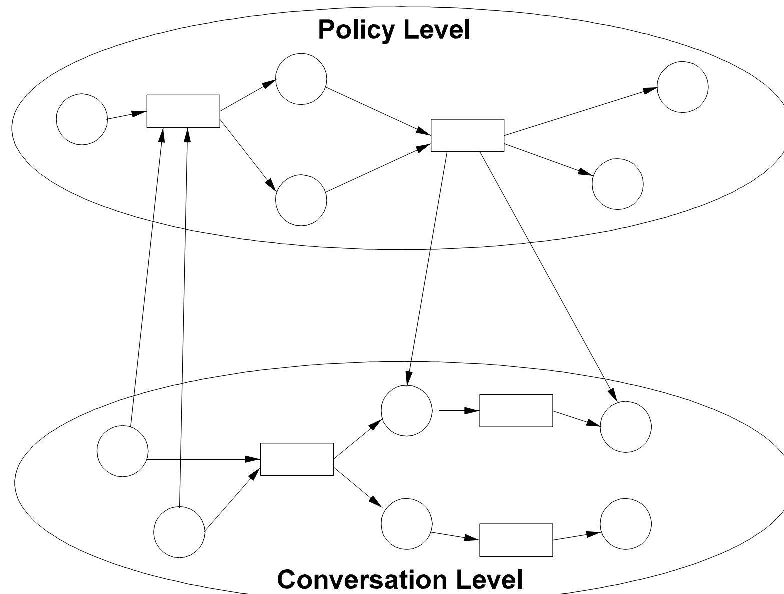

Figure 10. Petri net with conversation and policy levels.

It is natural to compose more complex conversation models out of simpler conversations or sets of protocols by connecting appropriate elements by arcs. It is important to note that complex conversations do not change the semantics of the protocols (subnets). For consistency, all basic act exchange schemas are defined via protocols, even if only a single communicative action is executed between two agents (single act without a response). That means that all communicative acts defined in an Agent Communication Language (ACL) (such as FIPA ACL or KQML have at least one protocol defined for them .

4.2 Conversation examples

FIPA has defined a collection of simple interaction protocols, which can be used in separation or in conjunction with other protocols. We first consider a simple FIPA interaction protocol, *request*. For an informal outline of the protocol, we have chosen a notation based on FIPA-97 specifications. FIPA used a notation (in its previous specifications) based on Deterministic Finite Automata, represented graphically simply as connected boxes. Boxes with double edges represent communicative actions, which can also be treated as states; white boxes represent actions performed by initiators; shaded boxes represent actions performed by other participants in the protocol. Connections between boxes can be interpreted as transitions. (For simplicity we have skipped `not-understood` responses, which can be sent in response to virtually any communicative act.) For the purposes of simplicity and clarity of the Petri net diagrams, we have only shown the names of places and transitions and have left the inscriptions, guards and marking unspecified.

4.2.1 FIPA request protocol

The FIPA request protocol simply allows one agent to request an action to be performed by another agent. The action request can be rejected or accepted, and once

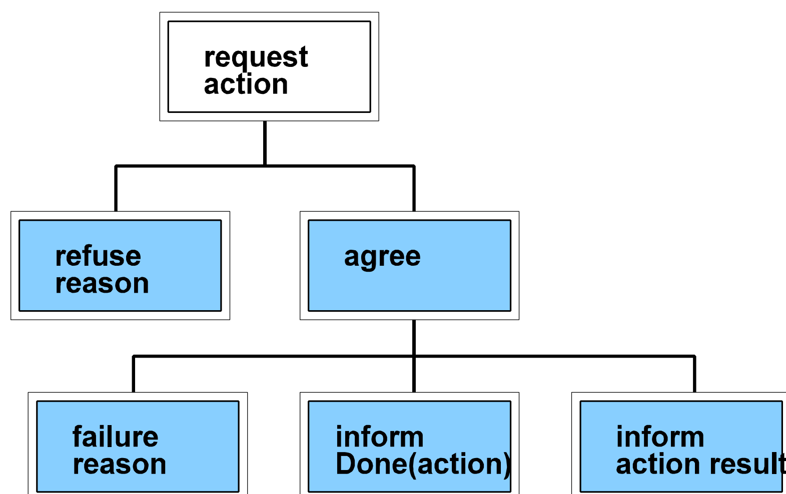

Figure 11. The FIPA request protocol.

accepted can be finished with a success or failure. The schematic representation of this protocol is shown on Figure 11.

The Petri-Net-based model of a simple request conversation is drawn in Figure 12. As discussed in Section 4.1 all the arrows (transitions) crossing the role boundaries represent message exchange between two agents (roles). We can call the upper role from the diagram, an *employer,* and the other role, a *contractor.* The conversation formally specifies where and how the interaction between two interested parties occur, and what communicative acts are allowed in particular stages of the conversation.

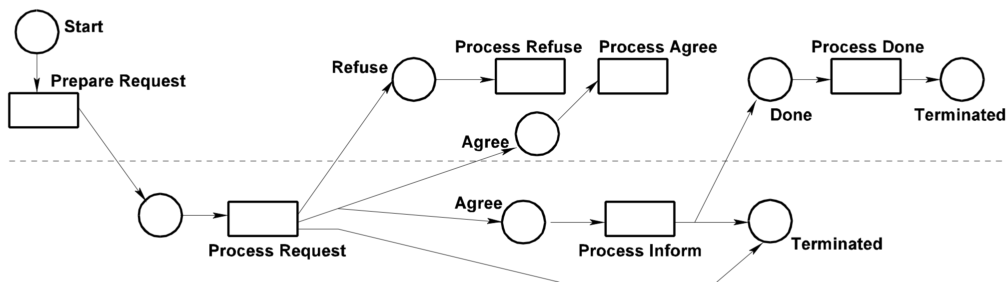

Figure 12. Petri net representation of the FIPA request protocol.

4.2.2 Contract-net protocol

We use here a modified version of the FIPA contract-net protocol, where the manager wishes a task to be performed by one or a group of agents according to some arbitrary function which characterises the task. The manager issues the *call for proposals,* i.e. the *cfp* act, and other interested agents can send proposals. In contrast to the original FIPA contract-net protocol, there is no need to do anything if an agent

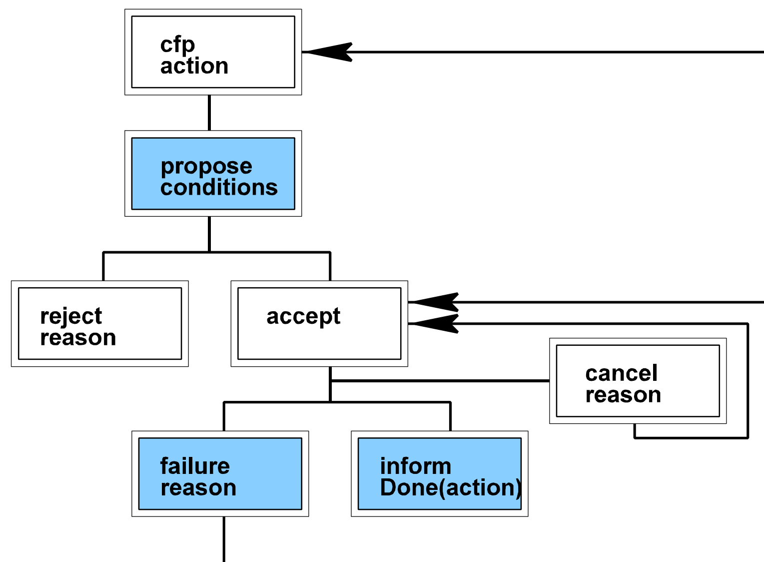

Figure 13. A custom contract-net protocol (FIPA notation).

playing a role of a potential contractor is not interested in submitting a proposal. That means that our contract-net model from the very beginning relies on the notion of timeout, i.e. some actions need to be performed in the event of a lack of enough proposals, or even in the case of a complete lack of proposals.

The proposals are collected by the manager, and then they are rejected or accepted. The accepted proposals can be cancelled, either by the manager via a *cancel* action, or by the contractor via a *failure* action. In case of cancellation, other submitted proposals can be reconsidered, or a completely new call for proposals can be issued. The schematic representation in the FIPA notation is presented in Figure 13, and the Petri Net model is shown in Figure 14. The contract-net initiator is a *manager* and all other participants are *contractors*. In the Petri Net case, we have depicted an example conversation based on the contract-net protocol between a *manager* and three *contractors*.

The actual behaviour of the manager and contractors is not specified by the example net shown: this information is encoded inside arc inscriptions and guards. Consider two potential strategies, i.e. conversation policies, the manager can follow during the course of the conversation:

1. Wait for the first two proposals, accept the best one, and reject the other one, and all other late proposals. If the chosen proposal fails, reissue the *cfp* again and follow this approach all over again until the task is successfully accomplished.
2. Wait for the first two proposals, accept the best one, but do not reject the second one – keep collecting incoming proposals instead. If the chosen one has finished successfully, reject all other proposals. If the chosen one fails, choose the next best, and iterate through the process until successful, or in the case of no more proposals waiting, reissue the *cfp*.

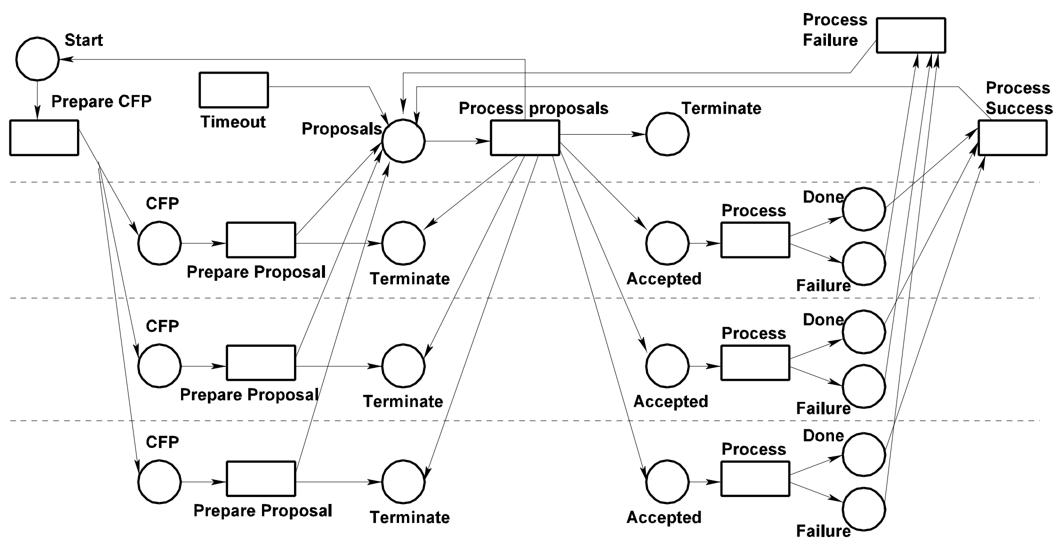

Figure 14. A contract net with three contractors in Petri net notation.

One can build more complex conversations based on the manager and contractor roles, and it is possible to combine two or more protocols into a single conversation model. It is also possible for a contract-net protocol to work together with request and inform protocols.

For the case of complex interaction schemas it is possible, but not necessarily desirable, to reuse the net models and structures for concurrent unrelated conversations, as the net is already being used in a concurrent fashion by a single concurrent conversation. (In such a case the creation of separate structures for each of the conversation instances would be suggested for the sake of simplicity, and this is the approach that we follow). With coloured Petri Nets, it is possible to use a single net structure even in those complex concurrent cases, but in such cases an appropriate additional matching based on conversation identifiers is necessary inside the arc expressions and transition code.

5. Deliberation and Task Scheduling

A high-level part of the Opal architecture is a set of standard agents to perform scheduling and planning for other agents. All micro-agents and ordinary agents are inherently goal-driven and role-oriented. The micro-agent platform built on top of micro-kernel provides set of standard agent services to perform hierarchical goal reasoning and planning, following Procedural Reasoning Systems [21] traditions. This is currently in the design phase and will be described in future publications.

Developers can use all the lower level machinery to perform simple task and goal decomposition and program appropriate scenarios to be used as a role implementations. Further, the developer can provide some meta-level reasoning agents and capabilities which will typically span most of the existing system components.

The principle idea is not to cope with the big task in a single centralized place (such as inside a large coarse-grained agent), but rather to reuse different bits and pieces distributed throughout the system. This is where the entire system can benefit from the micro-agent infrastructure. One of the main goals of Opal is to provide uniform modelling abstractions and operational techniques, which can be used for dealing with different scales and different granularity of components. A loosely coupled, but interconnected network of such components, an agent system, should be able to solve in a flexible and robust different complex systems.

6. Summary

This paper has described the Opal system which seeks to employ the notion of agent modelling at multiple levels of abstraction by using micro-agents. The micro-agent and supporting kernel implementations that we have developed in Java enable the software engineer to develop agent-based systems and components that are much more efficient than those developed by conventional coarse-grained agent technology. With Opal it is possible to design FIPA-based agent systems and also employ agent-based components for virtually all aspects of a software system, including finer-grained components that are not normally implemented in terms of agent constructs for reasons of efficiency. Opal also supplies an Agent Conversation Manager that incorporates the notion of higher-level 'policies' for guiding and constraining agent interactions. It is our contention that the approach and infrastructure described here supports a more scalable approach to agent-based solutions, because one can employ agent-based concepts over a wider range of software engineering activities and still produce efficient software implementations.

We are continuing to add more services and functionality to the Opal agent framework and will make the source code publicly available in the near future.

References

[1] G. Booch. *Object Oriented Analysis and Design with Applications*. Addison Wesley, 1994.

[2] N. R. Jennings. Agent-oriented software engineering. In *Proceedings of the 12th International Conference on Industrial and Engineering Applications of AI*, pages 410, 1999.

[3] N. R. Jennings and M. Wooldridge. Agent-oriented software engineering. In. J. Bradshaw, editor, *Handbook of Agent Technology*. AAAI/MIT Press, 2000.

[4] F. Bellifemine, A. Poggi, and G. Rimassa. Developing multi-agent system with FIPA-compliant agent framework. (31):103128, 2001. http://sharon.cselt.it/projects/jade.

[5] H. Nwana, D. Ndumu, L. Lee, and J. Collis. ZEUS: A Tool-Kit for Building Distributed Multi-Agent Systems. *Applied Artifical Intelligence Journal*, 13(1):129 186, 1999.

[6] S. Poslad, P. Buckle, and R. Hadingham. The FIPA-OS agent platform Open source for open standards. In *Proceedings of the 5th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents*, pages 355368, 2000.

[7] FIPA. FIPA ACL Message Structure Specification. X00061, http://www.fipa.org/specs/fipa00061/, 17 August 2000.

[8] M. Georgeff and A. S. Rao. A Profile of the Australian Artificial Insititute. *IEEE Expert*, pages 8992, December 1996.

[9] O. Gutknecht and J. Ferber. MadKit: Organizing heterogeneity with groups in

a platform for multiple multi-agent systems, December 1997. Technical Report 97188, LIRMM, 161, rue Ada - Montpellier - France. http://www.madkit.org.
[10] FIPA. Foundation For Intelligent Physical Agents (FIPA). FIPA 2000 specifications. http://www.fipa.org/specifications/index.html, 2000.
[11] J. R. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press: Cambridge, England, 1969.
[12] M. P. Singh. Agent communication languages: rethinking the principles. In *IEEE Computer*, 0018-9162, pages 4047. December 1998.
[13] J. B. Mark Greaves, Heather Holmback. What is a conversation policy? *Mathematics and Computing Technology*, The Boeing Company P.O. Box 3707, Seattle, WA, USA.
[14] T. Finin, Y. Labrou, and J. Mayfield. KQML as an agent communication language In *Software Agents*, 1997, also at http://www.cs.umbc.edu/kqml/papers/kqmlacl.pdf.
[15] R. Elio and A. Haddadi. On abstract task models and conversation policies. In Working Notes of the Workshop on Specifying and Implementing Conversation Policies, pages 89-98, May 1999.
[16] T. Wagner, B. Benyo, V. Lesser, and P. Xuan. Investigating interactions between agent conversations and agent control components. In Agents 99 Workshop on Conversation Policies. 1999.
[17] H. V. D. Parunak. Visualizing agent conversations: Using Enhanced Dooley graphs for agent design and analysis. In *Proceedings of the Second International conference on Multi-Agent Systems ICMAS'96*.
[18] J. Odell, H. V. D. Parunak, and B. Bauer. Extending UML for agents. In E. Y. Gerd Wagner, Yves Lesperance, editor, *Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, pp. 3-17, 2000. http://www.jamesodell.com/ExtendingUML.pdf.
[19] K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*, volume 1: Basic Concepts. Springer-Verlag, Berlin, 1992.
[20] S. Cost, Y. Chen, T. Finin, Y. Labrou, and Y. Peng. Using colored petri nets for conversation modeling, 1999. Available online at http://www.csee.umbc.edu/~jklabrou/publications/ijcai99acl.ps.
[21] F. Ingrand and M. Georgeff. *Procedural Reasoning System, User Guide*. 1991. Australian Artificial Intelligence Institute, 1 Grattan Street, Carlton, Victoria 3053, Australia.