Virtual Knowledge Communities for Distributed Knowledge Management:
A Multi-Agent-Based Approach using JADE

Diplomarbeit
von
Marc Hammond

Betreuer: Prof. Dr. J. Calmet
Betreuender Mitarbeiter: Dr. Pierre Maret

Institut für Algorithmen und Kognitive Systeme
Universität Karlsruhe (TH)
SS 2004
Abstract

Virtual communities are becoming increasingly popular, particularly on the internet, as a means for like-minded individuals to meet, share and gain access to the information they are most interested in quickly and efficiently, from other individuals they can learn to trust. The concept of a community of practice or a community of interest can be supported in a virtual community in order to bring the appropriate parties together, to share their knowledge with each other. In this project, the most general model for modelling distributed knowledge management is investigated, that involves extending the abstraction of an agent, such that it becomes an actor within a knowledge community. In this model, agents themselves, software or human, are the members of the virtual knowledge sharing communities. Agents can choose to join, leave, create and destroy communities, and can be member of many communities simultaneously. A Jade-based prototype design and implementation are used to show how such a system can operate and be useful in practice.

Project Website

There is also a website for the project where the prototype system can be obtained, together with this report, the user guide, and some examples. This can be found here:

www.doc.ic.ac.uk/~mrh00/individualproject/.
# Contents

1 Introduction ............................................ 2  
2 Background ............................................. 3  
   2.1 Motivation ........................................ 3  
   2.2 Related Works .................................... 4  
      2.2.1 Centralised Knowledge Management .......... 4  
      2.2.2 Distributed Knowledge Management .......... 4  
      2.2.3 Virtual Communities ......................... 6  
      2.2.4 Societies of Agents ......................... 6  
      2.2.5 The Role of Agents in Knowledge Management 7  
3 The approach taken to Knowledge Management using Virtual  
   Knowledge Communities .............................. 8  
   3.1 The 'Community of Communities' ............... 8  
   3.2 Agent Modelling .................................. 8  
   3.3 Community modelling ............................ 10  
   3.4 Agent Policies Regarding Community Life Cycle . 11  
4 Prototype System ...................................... 13  
   4.1 Aims the System ................................ 13  
   4.2 JADE ............................................ 14  
   4.3 Description of System ........................... 15  
   4.4 Design of the System ............................ 17  
      4.4.1 JAVA Classes ............................... 17  
      4.4.2 Agent Communications ....................... 22  
      4.4.3 Agent Behaviours ............................ 24  
   4.5 Implementation of the System .................. 30  
   4.6 Using the System ................................ 31  
5 Evaluation ............................................. 32  
   5.1 Example Use Case Showing Strengths of the System . 34  
6 Conclusions and Future Works .......................... 37
1 Introduction

The objective of this project is to investigate an agent based abstraction approach to distributed knowledge management called virtual knowledge communities, and to present a prototype implementation of the approach using JADE (Java Agent Development Framework) [4]. This paper begins by presenting the background to the project in Section 2. This explains why the research being conducted is important and useful, and introduces some of the concepts used in the project. State of the art projects, research and related work already done in the domain of distributed knowledge management have been researched and are also summarised in this section. Having studied and evaluated the existing work done in this field, the multi-agent system based approach to virtual knowledge communities is presented. This approach was conceived, by taking the strengths of other approaches and applying them to the most appropriate abstraction for distributed knowledge sharing in open systems. The approach is presented in Section 4. Next, a sample implementation that demonstrates how the approach can be put in to practice is presented in detail in Section 5. The implementation is then evaluated in Section 6 with regard to other knowledge management approaches, and conclusions are presented in Section 7.
2 Background

2.1 Motivation

The inception and popularisation of the internet has taken the information age to a new level of complexity. The information society now has so much information at its disposal that it is more important than ever to find effective techniques for managing the optimal distribution of this information, such that individuals are not overwhelmed with meaningless data. Knowledge exchange, knowledge brokering and knowledge reuse are of critical importance within the knowledge information society [1].

The objectives of knowledge management are well known; to improve the re-use of the knowledge within the processes of a system, by reducing the distance between tasks and generalised knowledge bases, and increasing accessibility to resources [1]. In recent years many organisations have adopted knowledge management techniques that focus on building large, expensive, centralized knowledge management systems based on the standardisation of the syntactical and semantic representations of all of the knowledge in the organisation. Support is then provided for tools that facilitate access and use of this knowledge by different users of the system. There is both practical and theoretical evidence to suggest that this approach to sharing knowledge is inappropriate, as the centralisation and standardisation of the knowledge removes much of the vital contextual information that inherently comes with knowledge from different sources [2].

Modern knowledge management, on the societal level, often focuses on the constitution of communities of practice and communities of interest [1]. This is particularly important in a large, heterogeneous system such as a company or the internet, where the ‘global’ community is too large to be useful as a single entity, without the formation of such smaller sub-communities that are composed of individuals with some common agenda.

The fact that the internet is itself a dynamic, open system, that is composed of distributed heterogeneous entities, who can both conflict and cooperate with each other but have no necessary central locus of control, makes the multi-agent system paradigm a convenient and natural abstraction for designing internet-based systems. This is also true of almost all corporate systems, in which knowledge needs to be shared. Actors within a society such as a company are both individuals and software systems, which can be generally abstracted as agents. Agents are themselves autonomous, and can be used to model human as well as automata behaviour, a requirement for modelling and engineering complex articulated systems.

Moreover, it is increasingly hard to explicitly define the role of the human and that of the software system in areas such as e-business task processing. An increasing amount of software is concerned with task and knowledge processing, and it is becoming increasingly common for programs to inter-
act and trade knowledge with each other without the direct influence of human control. Knowledge is no longer external to an agent’s processes; it is part of the ”agent’s world”, and knowledge management can be seen as ”result sharing” between agents. Agents possess knowledge and processes within the organization tend to make agents produce and exchange knowledge with each other. Result sharing is generally considered as a sub-part of the ”Distributed Problem Solving” area [3]. From the perspective of corporate knowledge management, agents cooperate to resolve a unique high-level problem, namely the success of a company.

There are numerous strategies for achieving this goal, thus agents may have no common explicit goal. In the context of this project, agents are considered to have the ability to process their own tasks or solve problems, which could gain in productivity, time or quality by importing knowledge from other agents. The aim of the project is to formulate an approach that enables agents to do this effectively without being forced to make concessions with regard to their own autonomy or corrupt the knowledge they have by imposing a standard ’view of the world’ upon them.

2.2 Related Works

2.2.1 Centralised Knowledge Management

Most approaches to knowledge management - even when based on MAS - remain predominantly founded on centralization and objectivity. Examples of such systems are numerous, for instance [12] and [9]. However, this approach appears incompatible with the very nature of knowledge.

Bonifacio [5] criticises most current knowledge management systems, in which ”all perspectival aspects of knowledge should be eliminated in favor of an objective and general representation of knowledge”. It is argued that many traditional knowledge management systems are deserted by their users due to the ”false epistemological and organisational assumptions made during the design of the systems”. This is unacceptable in a modern knowledge sharing environment.

2.2.2 Distributed Knowledge Management

Kornfeld [13] claimed years ago that diversity and concurrency of (scientific) communities are essential to their progress. The same argument can be applied to any community in which knowledge is the currency.

Distributed Knowledge Management is presented in [2], emphasising the value of autonomy and coordination within an approach the authors have called Distributed Knowledge Management. It is based upon two fundamental principles; the Principle of Autonomy, which states that each organisational unit should be granted a high degree of autonomy, and in particular semantic autonomy, to manage its local knowledge and conceptualisations,
The ‘Knowledge Node’ approach

The approach taken in this paper

KEY: □ = agent  ○ = organisational unit

Figure 1: A comparison of the view of ‘organisational units’ in the approach taken in this paper and the knowledge nodes approach described in [2].

and the Principle of Coordination, which states that each unit must be enabled to exchange knowledge with other units, without imposing the adoption of a single, common interpretative schema, but by using a method of mapping other unit’s context onto its context from its own perspective.

A Knowledge Node (KN) is then presented as a means for obeying these principles and is defined as a reified organisational unit which exhibits some degree of semantic autonomy. Each KN has its own explicit representation of its reified perspective of a community called a context. This can be a category system, an ontology, a collection of guidelines, or a business process. Each KN also has a software agent associated to it, which knows the KN’s context. The agents act as a go-between between the KN and other KN’s, by supporting users of a KN to compose outgoing queries, and answering incoming queries from other KN’s.

The principle of autonomy is supported in this project, but it is felt that the principle is so strong that the idea of using Knowledge Nodes is not sufficient. The ‘organisational unit’ mentioned in the above principle should go right the way down to the level of the agents themselves. Since every single agent, even those in the same community, needs its own autonomy in order to model an organisation and its individuals correctly, then this is the natural model for an agent based knowledge sharing system. With this model, agents can choose to join, leave, create and destroy communities, and can be member of many communities simultaneously. This is the most general model of a fully autonomous system, and the difference between this model and that proposed in [2] is portrayed in Fig1.
2.2.3 Virtual Communities

Virtual communities are becoming increasingly popular, particularly on the internet, as a means for like-minded individuals to meet, share and gain access to the information they are most interested in quickly and efficiently, from other individuals they can learn to trust. An example of a system that implements such a community can be found in [6]. The concept of a community of practice or a community of interest ([2] and [6]) can be supported in a virtual community in order to bring the appropriate parties together, to share their knowledge with each other. An advantage of this is that the members of a community centred on one specific topic or practice will only be presented with knowledge from domains they are, or at least are relatively likely to be, interested in, and this knowledge need not be something they have specifically asked/searched for. Sometimes a piece of information may be of interest to an agent, although it didn’t know that it was looking for it. While many virtual communities already exist on the internet to aid knowledge sharing, many of which use ‘agents’ in various forms as part of the system, we propose a model that extends the abstraction of an agent, such that it becomes an actor within the system. In other words, in our model, agents themselves, software or human, are the members of the virtual knowledge sharing communities.

2.2.4 Societies of Agents

Within multi-agent societies a balanced articulation must be found between organizational control and autonomous social behavior of agents. Bradshaw proposes a framework to specify, manage and enforce agent behavior using DAML-based policies [15]. Exploring security issues inside an open organisation, Omicini ([16]) claims for a systemic vision of MAS, explicitly accounting for social issues (social intelligence) as opposed to focusing on individual agent’s intelligence.

Calmet presents the Liberal Approach to openness in societies of agents in [7]. This approach is based on Max Weber’s ideas ([14]), that the behaviour of the society is the result of the individual actions of the agents. The paper states that “the goals of a multiagent system are the results of specific goals of the constituting agents which are closely tied to the design decision adopted when defining the architecture of the system”. The work undertaken in this project fully subscribes to this view of achieving goals within the system, because, as already argued above, the agents within any knowledge sharing corporation will not naturally subscribe to a centralized model of control and standardization. Therefore the most general model that can be applied to the problem of corporate knowledge sharing is one in which each agent has full decision making autonomy and his own perspective of the world, within some sensible general laws and boundaries, which
can be imposed for reasons such as security of the overall system.

### 2.2.5 The Role of Agents in Knowledge Management

There are two general perspectives of the roles of agents in knowledge management [8]. The first is that agents implement the functionality of the system, as, for example, in [9]. The other is that agents are used to model the organisational environment in which Knowledge management will operate. The approach presented in this project is perhaps closer to this second perspective; however we opine that choosing the correct approach for modelling the real world is the best way of designing a system that will work well in practice. In other words, although we are modelling an organisational environment with agents, the agents in this model can also be used to create a fully functional system, for example, one in which agents play the part of a personal assistant, which searches for data on behalf of its human owner.
3 The approach taken to Knowledge Management using Virtual Knowledge Communities

Having analysed the existing body of work in the area, we propose an autonomous multi agent system based on the virtual knowledge community abstraction, which adheres to the liberal approach to societies of agents.

3.1 The 'Community of Communities'

A community is a place where agents can meet and share knowledge with other agents who share a similar domain of interest. We propose a 'community of communities' of which all agents are a member, and within this global community, communities can be created and destroyed dynamically by agents, as and when necessary.

As with any such open system, consisting of distributed individuals with different views of the world, we have the problem of knowledge heterogeneity. This is encapsulated by the buyer seller paradigm in [10]. There needs to be a way to overcome this problem so that agents can communicate and share ideas about topics, and understand what each other mean when they use concepts to describe these ideas. There have been some developments in solving this problem, in [11] and [10], but this problem is considered outside of the scope of this project.

In order to address this problem we assume that, even if agents are fully free to maintain and keep their own ontological view of the world, we have a set of normalised ontologies, which all agents agree upon and can reference when creating communities. Here, an ontology is a taxonomical specification of all the classes of objects and predicates that an agent understands. It is used to give meaning to instances of the objects and predicates described within it, that an agent may encounter. Thus, other agents can understand what the domain of a community it in the context of this normalised ontology. It is important to note that an agent’s personal knowledge is not constrained to solely contain concepts from the normalised ontology. If this were the case, the virtual knowledge communities approach would be conceptually no more powerful than a centralised one.

The community of communities maintains a central directory of all communities, which agents can use to find out about existing communities they may wish to join. All agents in the organisation have access to this directory.

3.2 Agent Modelling

In the perspective of this project, agents are active objects with the ability to perceive, to reason and to act. In addition, it is assumed that agents have explicit knowledge representation and communication ability [17].
Participation in knowledge communities does not replace the intrinsic activity of agents for which they were introduced to the agent society. Thus, agents follow two goals: their intrinsic goal, which they may operate without the community help, and knowledge acquisition goal, which is processed, proactively or reactively, for improving intrinsic activity. This project considers only the knowledge acquisition component of an agent, but respects the fact that that is not the only goal of the agent.

We have four key notions for our agent modelling approach: personal ontology, knowledge instances, knowledge cluster and mapper. These are explained hereafter.

The knowledge of an agent is represented in the vocabulary of its personal ontology, which describes the taxonomical relationships between classes of concepts and predicates it understands. Knowledge instances are instance of objects defined into the personal ontology. We assume that an agent’s knowledge consists of both its personal ontology and knowledge instances of objects in its personal ontology.

Under this assumption, an agent’s knowledge varies from agents to agent, which is fully compliant with individuals’ knowledge. Moreover, while processing tasks, agents use, produce and acquire knowledge. Thus, knowledge can not be uniquely considered at design time (inherent knowledge). So, we assume that agent’s knowledge evolves during the agent’s life. These assumptions are trivial regarding knowledge instances, but they are not trivial regarding its personal ontology.

In order to maintain an evolving personal ontology, and share its knowledge with other agents an agent must have a means of adding and taking concepts and predicates to and from its personal ontology. In order to do this we introduce the concept of a knowledge cluster to define a sub-part of an ontology, in other words a ‘piece’ of knowledge, that can be shared between agents.

A knowledge cluster is defined by its head concept. It contains the head concept itself, all sub-concepts of the head concept, and all concepts that are the types of attributes of the head concept and its sub-concepts. It also includes all predicates that only take arguments whose types are one of the concepts in the knowledge cluster.

Basic operators on knowledge clusters can be defined. Knowledge clusters can be added together, compared for equality and extracted from other knowledge clusters as sub clusters defined by a head concept. The overlap between two knowledge clusters can also be found.

The personal ontology of an agent can also be represented by a knowledge cluster, without a defined head, but instead containing all concepts and predicates known to the agent. Using this fact and the basic operators on knowledge clusters, we are able to extract knowledge cluster from the personal ontology of an agent in order to share it with another agent, and add knowledge for personal use.
While accessing knowledge of a community, an agent can choose to add a knowledge cluster to its own ontology and then use it for “private” purposes. Contributions to the community are similarly related to the community’s cluster: the agent extracts a cluster from its ontology and send it to the community.

The only barrier to this sharing of knowledge is the heterogeneity of knowledge mentioned above. Therefore, for each community that an agent is a part of, the agent must maintain a set of mappings from its own personal ontology concepts and predicates to those it encounters within the community. This includes the maintenance of mappings from its personal ontology concepts and predicates to those in the normalised ontology. In order to do this we introduce the concept of a mapper.

The mapper simply contains a set of mappings from personal terms to a mapped terms, and allows an agent to add such mappings, and use the mapper to normalise or personalise a given knowledge cluster or knowledge instance using these mappings. An agent will maintain one mapper for each community it is a member of, and mappings can be added as the agent sees fit. Once a mapping is added to the mapper, this mapping will always be used when normalising or personalising a knowledge cluster or a knowledge instance. Note that mappings to normalised ontology concepts and predicates can exist outside of a community, and span more than one community. Therefore, any mappings to normalised ontologies are stored by the agent outside the context of any community. They are, if you like, the agent’s global mappings for a given normalised ontology.

3.3 Community modelling

Each virtual knowledge community has a *domain of interest*, which is similar to the concept of ontology for an agent. We will consider this domain of a knowledge community as a knowledge cluster, which is given by the community leader who created the community. The leader of a community is an agent who controls knowledge exchanges within the community, and has power to destroy the community. There can be more than one leader of each community. Other agents can become the leader of a community if the current agent wishes to leave the community, or if the current leader is too busy and an extra leader is needed.

The *community pack* is what defines the community. It consists of a community knowledge cluster, a normalised ontology which contains at least the head of the community cluster, and the identification of the leaders of the community. It is created when the community is created, and lasts for the lifetime of a community.

In order to share knowledge within a community, a blackboard based system was considered. Blackboards are repositories for storing agents’ multiple intermediate results [3]. Our goal is compatible with blackboard systems.
However, our aim is not to generate a huge central knowledge base, which would be unpractical. We prefer the use of a buffer system where messages are stored and then deleted. The oldest, lowest priority message is deleted first. In addition, blackboard architectures are generally used for agents cooperating to solve a problem. Our perspective is different because we do not consider a unique system goal: agents cooperate to solve their respective problem.

Thus, knowledge is shared by the members of a community using messages that are written to the community buffer. Messages are stored here and can be read by other members of the community. This ensures that agents do not need to know which other agents are members of the community, but can still access knowledge in the community, and themselves choose whether it is interesting or not. This is a fully autonomous approach, in which the individual actors in the system have the power to decide if the knowledge is interesting to them or not, as opposed to the large centralised approach where such decisions are imposed from above.

In general terms, each community must have a policy for specifying who is able to join the community, whether or not agents within the community have roles, and what these roles should be and mean, and whether or not there are any additional rules corresponding to how much, or how often an agent must contribute to a community. The leader of the community is in charge of enforcing these policies within the community.

Since the communities in our system are based on the liberal approach to a society, what actually defines a community is contained within the constituent agents that make up that community. Moreover, all communities are created by agents themselves, so any general policies for a given community are also chosen by the individual agent who creates the community.

Every agent can be in any number of communities at any one time. An agent may be interested in a community when the intersection between the community’s cluster and its own ontology is not empty. In order to find this intersection, the agent will use the mappings it maintains between its personal ontology, and those of the normalised ontology of the community, in order to find an overlap between its own personal ontology and the domain knowledge cluster of the community.

3.4 Agent Policies Regarding Community Life Cycle

Agents create communities. Therefore agents need a policy specifying what are the circumstances in which they would choose to create a community are. Of course, this policy can be different for every agent, and can range from creating a community for every domain of interest the agent has, purely for the good of all, to creating communities purely because the agent needs to in order to get some knowledge on a given domain from other agents.

We also need a policy for defining when an agent would join a community.
Again, each agent can have its own policy for joining communities and this can range from joining every community, whose domain of interest overlaps with the agent's personal knowledge, to only joining communities in which the agent feels it can benefit, because it needs to get some knowledge from a given domain, and there already exists a community that concerns this domain of interest.

The circumstances in which an agent leaves a community also need to be defined. An agent could conceivably be forced out of a community, or choose to leave. The circumstances in which an agent would be forced out of a community are defined by the policies of the community that specify the minimum contribution of an agent, mentioned above. The circumstances in which an agent would choose to leave a community are of course down to the individual agent. Policies range from leaving as soon as the agent gets what it needs from a community or if the agent feels it is not benefiting from being in the community, to never leaving a community, unless forced to.

Finally a community can die for several reasons. Policies range from leaders killing the community once they have what they need from it, to communities being killed due to a lack of activity e.g. once everyone has left the community, or no messages being exchanged for a certain minimum time period.

All of these policies are policies of the individual agents, as both leaders and non-leaders. All of the policies mentioned range from totally selfish agents, who do everything for the good of themselves, to totally self-less agents, who do everything purely in the hope that it will help others, i.e. for the good of the group. Of course, an agent’s policy for any given action can fall in between these two extremes.
4 Prototype System

4.1 Aims the System

The aim of the prototype is to design and create a working system, in which agents with heterogeneous personal ontologies can create, destroy, leave and join communities in order to share knowledge clusters and instances with each other, as described in the previous section. The intention is just to get a system working, with two extreme different 'types' of agents. The prototype system will prove that such a system can be built and how different types of agents can be implemented. It will also show how such a system can be useful, in that it will be a working system that does what it is meant to, namely allow the sharing of both heterogeneous knowledge clusters and instances in a distributed environment.

The prototype system is not intended as a final product. The user interface can be primitive, and not all feasible policies and agent decision making mechanisms will be implemented. Since this product forms a part of a research initiative, the emphasis is not on making a perfect product, rather, to create an initial version of something that shows it can work, and to put the data structures and classes in place, such that the implementation can be extended and developed further should the need and desire arise. The emphasis of the system is on the sharing of knowledge, the creation of mappings between personal concepts and community/normalised concepts and to establish the mechanics with which agents can dynamically move between communities and update their personal knowledge.

All agent decision making processes that are not considered within the scope of the prototype will be implemented in the prototype by asking a human user to make a decision on the agent’s behalf. This is an appropriate model for decision making to take in light of the agent abstraction methods we are aiming at, as we have already stated that an agent in a system could in fact be a human anyway. The idea is that should an appropriate automated decision making process be devised for a certain type of agent in the future, the current human interaction can be replaced by the new decision making process in the implementation of the new type of agent.

So to summarise what is expected of a prototype we list the inputs to the system, the decisions we expect a human user to make on the agents behalf, and the results we can expect to observe.

We expect the prototype to allow the creation of multiple agents, potentially on different machines, each with a given personal ontology, given personal knowledge instances, a set of given normalised ontologies and a set of mappings for each normalised ontology as inputs. Each one of the sets of mappings details any mappings an agent already has from its personal ontology predicates and concepts to the normalised ones. The agents should be allowed to join and leave the system at any given time, in any given order.
The agents should not be hard-coded with any knowledge about existing communities to join (except for the global ‘community of communities’) or other agents to communicate with. It should search for these things itself by using the community of communities.

We expect the user to be involved when an agent creates a new community. Here the user should specify which one of the normalised ontologies the agent should use to create this community. We also expect the user to be involved whenever the agent encounters a new concept that it does not already have a mapping for. Here the user is asked which concept to map it to, if any. Finally, we expect the user to be involved when adding goals to an agent (for goal driven agents) or when choosing which of its knowledge it should share (for altruistic agents). For all of these types of decisions, we could imagine an automated approach, but this is outside of the scope of the prototype implementation of the system described in this project.

The results of the system should take the form of a user readable interactive output, showing the actions the agent is undertaking as it does them, so the user can analyse the behaviour of the agents. This is very useful, as the actions the agents take depend on other agents in the system as well as themselves, so the actions taken by an agent may be non predictable in advance.

Also, agents should update their personal ontology, personal knowledge instances and personal mappings when appropriate, and these changes should be output to some concrete representation of knowledge in a standard format, that can be viewed and used by another human or software user.

4.2 JADE

JADE (Java Agent DEvelopment Framework), a Java based software development framework that fully conforms to FIPA standards for intelligent agents ([18]), was chosen to implement the prototype system. The fact that JADE conforms to all the appropriate FIPA standards and enables portable and easily maintainable agent development using the Java language, a language the author is very familiar with, makes it a perfect framework in which to develop a multi agent based system for knowledge management using virtual knowledge communities. Hereafter we detail the features of JADE that make it the ideal framework for developing the prototype system.

JADE allows the development of agents in java by simply extending, implementing using its collection of classes and interfaces, that provide a wide range of agent functionalities that are required to build a multi agent system. It also provides a FIPA compliant agent platform, upon which agents can be deployed. The platform can be split between several hosts, and provides an Agent Communication Channel (ACC) which controls all of the message exchanges within a platform, and to/from remote platforms.
This means JADE allows agents across distributed machines to be deployed into the same system and communicate with each other. This is one of the requirements of the prototype system.

Agents are created by simply extending the `jade.core.Agent` class. This makes the development of different kinds of agents very straightforward, and all of the fundamental things that an agent needs at its disposal are available to the programmer.

Several abstract agent behaviour classes are provided, which can be extended to provide different behaviours for different types of agents as required. Basic behaviours already provided by JADE include cyclic behaviours and one shot behaviours. Composite behaviours are also provided by JADE, with the scheduling of sub-behaviours handled differently depending on which behaviour is used, for example the programmer can choose to implement parallel behaviours (scheduled in a round-robin fashion) or sequential behaviours (scheduled sequentially), and add them to his agent class. This takes much of the complex consideration of how to handle the multi-tasking within one agent away from the programmer, and again, allows the programmer to focus on developing the behaviour it needs. This is ideal for the prototype system, as the different types of agents in the system need to have different types of behaviour, some of which are comprised of multiple behaviours.

Perhaps the most important aspect of JADE that makes it useful for the prototype system is its support for message passing and user defined ontologies. JADE provides the mechanisms for ACL Message passing as described by the FIPA specifications ([18]). It provides codecs and ontologies for converting java objects to and from the string representations that are sent in messages. All ACL Message performatives are provided, and the programmer can define his own ontologies, so everything is at the programmer’s disposal to allow the filling the content of messages appropriately for sharing knowledge.

No other frameworks were seriously considered, as JADE provided everything necessary in order to implement the prototype system required, and allowed the author to do the agent development in JAVA, which would always be the language of choice. Of course, the fact that JADE is an agent based platform, means it can be built and agents deployed on any platform, increasing the value of the prototype system, if it were ever to be developed further into a finished product, particularly when one considers the emphasis on openness and accessibility to all in the distributive system we are building.

4.3 Description of System

We have designed a prototype system that supports different ‘types’ of agents and communities with different ‘policies’, in order to enable us to
compare different types of policies and types of agents. We have only de-
dsigned a system to demonstrate communities with the basic extreme policies,
using two fundamental types of agents: the ‘individualistic agent’ and the
‘sociable agent’. However, the system could relatively easily be extended to
support a larger range of policies for each agent.

We define four policies that define a community. The Membership Policy
defines what type of agents can join the community. If a community has
an individualistic approach, then only individualistic agents can join the
community. If a community has a collective approach, then only sociable
agents can join the community. If a community has a free policy, then
any type of agent can join the community. The Regulation Policy defines
whether or not we have roles within the community. If there are roles then
the Role Policy defines the rules by which an agent obtains a ‘better’ role.

The different roles that exists are contributors, who can only send informs
to the buffer, full members, who can send informs or requests to the buffer,
and leaders, who can do the same as full members, but have additional
special duties. One approach for the Role Policy is a reward approach based
on quantity of knowledge submitted. In this approach an agent’s role is
updated if they have sent enough contributions to the community. The alternative is a reward approach based on quality of knowledge submitted.
This policy requires some form of feedback from the knowledge receiver.
Finally, the Contribution Policy of a community defines what the rules are
for agents to be part of a community, in terms of minimum contributions.
Either there is a minimum contribution per request for each agent, or a
requirement for a regular contribution in terms of time.

It must be noted that agents themselves create communities, so all of
these community policies are in fact community creation policies of the agent
who created the community. For this reason, all agents have parameters
specifying their community creation policies, as well as their individual agent
policies. In our prototype implementation we say an agent can be either
‘individualistic’ or ‘sociable’. This defines in which situations an agent would
create, destroy, join and leave a community. When an agent is created we
specify the class of agent it is (either individualistic or sociable).

An individualistic agent instigates or joins a community because it needs
knowledge from a certain domain, and will leave a community when it has
got what it wants from the community or if it is not getting enough from it
(e.g. it has been in the community too long with no benefit). An individu-
alistic leader will kill a community once it has the knowledge it requires.

A sociable agent instigates a community purely because it has nothing
else to do. The community will only die if everyone leaves the community
(community must have had at least one additional member join and leave
before this happens otherwise it would die as soon as it was created). So-
ciable agents join every community that concern them (whose community
cluster overlaps his own domain of knowledge) and never leave active com-

16
Figure 2: UML Class diagram showing the types of agent classes in the prototype multi agent virtual knowledge communities system.

4.4 Design of the System

4.4.1 JAVA Classes

All agents within the prototype system extend the jade.core.Agent class. The different types of agent class in the system can be seen in Fig.2.

CommunityOfCommunitiesAgent The CommunityOfCommunitiesAgent class represents a special agent, that has been created for conveniance in this implementation. The motivation for this agent type is as follows. All agents within the system need to have access to a shared CommunityOfCommunities class. The idea of creating of a static shared community of communities class, using the Singleton design pattern [20] immediately springs to mind. However, this approach would not work as agents are distributed objects, and each agent has its own memory space, so they would not be able to access the same static object. Therefore another approach is necessary. The approach taken here is designed to be as simple as possible, and is that we have a CommunityOfCommunitiesAgent that simply holds a local copy of the latest version of the CommunityOfCommunities class, via which other agents can access and update as and when necessary, by sending and receiving messages to it.

An alternative approach would be to use a data file accessible to all agents, but then we would have to handle issues of concurrent access, which are already taken care of by using an agent that holds the data item for us. It is not important that agents read the most up to date version of the
CommunityOfCommunities object, but it is important that any writes to this object (e.g. addition of new communities into the community of communities, or the removal of communities from the community of communities), are performed independently and sequentially, ensuring that one change is not overwritten by another. Using the single CommunityOfCommunitiesAgent to handle accesses to the object will ensure this.

CommunityAgent The CommunityAgent class is an abstract class that all different types of agents (except the special CommunityOfCommunitiesAgent) must extend. In this design we propose two extensions of the CommunityAgent; the IndividualAgent, and the SocialAgent. These two different types of agent have different behaviours as will be detailed later.

The classes that the CommunityAgent has access to can be seen in Fig.3.

AgentViewOfCommunityOfCommunities The AgentViewOfCommunityOfCommunities class is an interface for the CommunityAgent to access the CommunityOfCommunities object indirectly. Every CommunityAgent has exactly one of these. All accesses and updates to the community of communities are done through this interface, which sends and receives messages to the CommunityOfCommunitiesAgent on the CommunityAgent’s behalf. It is useful to encapsulate these accesses in this class, so that if, in any time in the future, it is decided to use a different means of handling the global CommunityOfCommunities object, the CommunityAgent class does not need to be changed. Instead, the methods in the AgentViewOfCommunityOfCommunities will just need to be changed.
Each CommunityAgent also has exactly one PersonalKnowledge object, representing its personal knowledge. The personal knowledge of a CommunityAgent consists of both its personal ontology and its personal knowledge instances. The PersonalKnowledge object is used by the CommunityAgent in all situations where its knowledge needs to be used or added to. It allows the agent to interpret new knowledge instances using the personal ontology of the agent, and provides methods to add knowledge clusters or knowledge instances to the personal knowledge of the agent, to extract knowledge clusters or knowledge instances from the personal knowledge of the agent, to check whether the agent already has a knowledge clusters or knowledge instances, and to get the overlap of the agent’s personal ontology and a given knowledge cluster. It also provides methods for importing and exporting the current personal ontology and personal instances to a data file, which was a requirement of the prototype system. The class diagram of the PersonalKnowledge class can be seen in Fig. 4.

Figure 4 shows that the PersonalKnowledge class extends the KnowledgeClusterOntologyClass class, which is in turn extends the jade.content.onto.Ontology class. This means that the PersonalKnowledge class is an ontology itself. The reason for this is that, in order to interpret knowledge instance messages, which contain content of the type jade.content.abs.AbsContentElementList (as will be detailed later), the agent has to have an ontology that is able to comprehend such content. JADE provides a means to do this, with the use of the jade.content.onto.Ontology class. This class, and classes that extend it, can be used, along with a codec, to convert the content of a message to/from a java object that an agent can use. Since the personal ontology of an agent is updated throughout the agent’s lifetime, and is therefore not constant or known at compile time, the agent needs an object that extends the jade.content.onto.Ontology class that represents its current personal ontology for understanding.
jade.content.abs.AbsContentElementList knowledge instances. The PersonalKnowledge class provides that object.

Moreover, as can be seen in Fig.4, the PersonalKnowledge class also has a reference to exactly one KnowledgeCluster, called its personalOntology, and exactly one AbsContentElementList, called its absInstances. It appears strange that we have a KnowledgeCluster representation of the personal ontology, when we have just stated that the PersonalKnowledge IS the personal ontology. The truth is that the current personal ontology is maintained in both formats. As already stated, the PersonalKnowledge representation is used to interpret the jade.content.abs.AbsContentElementList knowledge instances that the agent receives in messages and to fill messages with jade.content.abs.AbsContentElementList knowledge instances. The KnowledgeCluster representation is used for all other activities; mapping the personal ontology using a mapper, taking sub-clusters from the personal ontology, checking if a knowledge cluster overlaps the personal ontology etc.

The agent’s absInstances, of type jade.content.abs.AbsContentElementList, are the knowledge instances of predicates and concepts that appear in the agent’s personal ontology, that the agent currently holds. The PersonalKnowledge class also provides methods for importing and exporting these to file.

KnowledgeClusterOntology The KnowledgeClusterOntology is an ontology that describes the classes of objects that make up knowledge clusters, namely concepts and predicates. One might say that it is a meta-ontology that describes knowledge clusters. An agent’s PersonalKnowledge extends this, and in doing so can be used to fill the content of an ACLMessage with a KnowledgeCluster.

MemberViewOfCommunity The MemberViewOfCommunity is the CommunityAgent’s view of a community, and provides methods that an agent needs for a community. A CommunityAgent has exactly one MemberViewOfCommunity for each community that it is a member of. The MemberViewOfCommunity class diagram can be seen in Fig.5.

CommunityPack The CommunityPack contains all the information that is needed by a CommunityAgent with regards to a community it is a member of. It has a reference to the communities’ domain cluster, which is a KnowledgeCluster that represents the knowledge domain of a community, and a NormalisedOntology. The community pack is created by the CommunityAgent that creates the community, and lasts for the life-time of the community. Upon creation of a community, the creator selects a NormalisedOntology for the community, and a domain cluster is extracted from the creator’s own PersonalKnowledge, whose head concept is con-
Figure 5: UML association diagram showing the MemberViewOfCommunity and LeadersViewOfCommunity classes.

Obtained in the NormalisedOntology. Every agent who joins a community is given a reference to the CommunityPack for the community.

**NormalisedOntology** A NormalisedOntology object is constructed using its name and the NormalisedOntology data is imported from an .rdf file with the same name. It represents a normalised ontology that all agent’s in the system agree upon, and since all community domain clusters must have a head concept contained within a chosen NormalisedOntology for the community, it ensures that all CommunityAgents can understand the domain of a community, and decide whether or not to join it.

**Mapper** The Mapper class provides an object that a CommunityAgent can use to map its own personal concepts and predicates to those it encounters in a community. Using this mapper, agents can personalise KnowledgeClusters they encounter, in order to understand them in terms of their PersonalKnowledge, or normalise personal KnowledgeClusters in order to share them with the community. A CommunityAgent maintains one mapper for each community it is a member of.

The Mapper class also provides methods to import and export mappings to/from NormalisedOntologys to file. The mappings contained in these files will thus span all communities that the agent is a member of. This is the right scope for these mappings, since they are mappings to global normalised concepts.

**LeadersViewOfCommunity** The LeadersViewOfCommunity is the CommunityAgent’s view of a community that it is a member of, seen from
the point of view of the leader. It provides methods that a leader needs for a community. A CommunityAgent has exactly one LeadersViewOfCommunity for each community that it is a leader of, and will also still maintain a MemberViewOfCommunity. In other words, a leader of a community is also a member of a community.

CommunityBuffer The LeadersViewOfCommunity has a reference to exactly one CommunityBuffer for the community. This object is used to store and allow retrieval of ACLMessage's, by the leader. All messages within a community are sent via the leader of the community, who maintains a list of jade.core.AID's for each CommunityAgent member of the community.

4.4.2 Agent Communications

All communication in the system is done using message passing. CommunityAgents write knowledge clusters and knowledge instances, and requests for knowledge clusters and knowledge instances, by sending messages to the leader of the community, who adds them to the CommunityBuffer. CommunityAgents read knowledge clusters and knowledge instances, and requests for knowledge clusters and knowledge instances, by sending read request messages to the leader of the community, who then extracts messages from the CommunityBuffer, and sends them to the appropriate agent.

Leaders are in charge of the CommunityBuffer, and all communication goes via the leader. This is due to the fact that it is the leader who upholds any regulation and role policies that exist, by ensuring that any agent who sends a message to the community is actually allowed to do so. If the agent is not allowed, then the message is ignored.

Since knowledge is the object of the agent communications, we propose to define a high-level ontology for frame-based description of knowledge. We describe knowledge in terms of Concepts and Predicates, which have slots and arguments respectively, in an ontology we have called the KnowledgeClusterOntology. Using this ontology, and using a codec, JADE allows us to convert a KnowledgeCluster java object into an encoded String to be sent in the content of a message.

The KnowledgeSharingOntology extends the KnowledgeClusterOntology, and also defines the following basic actions and predicates that agents can send as the content of message. The following actions are implementations of the jade.content.AgentAction class; joinCommunity, readFromBuffer and updateRole. The following predicates are implementations of jade.content.Predicate class; communityDead and leavingCommunity. The KnowledgeClusterOntology is a static ontology that all agents can access using the static KnowledgeClusterOntology.getInstance() method.
To summarise, all message sent are of type `jade.lang.acl.ACLMessage`. They are sent with performative set to `jade.lang.acl.ACLMessage.INFORM` or `jade.lang.acl.ACLMessage.REQUEST`. All messages have the user defined parameter `CommunityAgent.COMM_ID_PARAM` with value of the community ID of the relevant community concerned. This is necessary, as agents can be members of several communities simultaneously, and need to know which community any message they receive concerns.

The user defined parameter `CommunityAgent.BUFFER_TIME_STAMP` is also used for `inform` and `request` knowledge messages, that the leader sends from the buffer. The leader adds this value as it sends it to a member, so the member knows the time of the last message it read from the buffer, and in the future can ask for later ones.

### Request messages

There are three types of `request` messages that can be sent by `CommunityAgent`'s.

The first is the `request` message containing the abstract agent action `(AbsAgentAction) joinCommunity`. This represents a request from a `CommunityAgent` who wishes to join a community, and is always sent to a leader of a community. This action takes an integer parameter `communityID`, specifying the community that the agent wishes to join. For this type of request, the `ontology` slot is set to `KnowledgeClusterOntology.getInstance()`.

The next is the `request` message containing the abstract agent action `(AbsAgentAction) readFromBuffer`. This represents a request from a `CommunityAgent` to read from the `CommunityBuffer` of the community, and is always sent to a leader of a community. This action takes an integer parameter `time`, specifying that the agent wishes to read the first message on the buffer after time `time`. For this type of request, the `ontology` slot is set to `KnowledgeClusterOntology.getInstance()`.

The final type of `request` just contains a String, that represents the head of a `KnowledgeCluster` that an agent wishes to increase its knowledge (both ontological knowledge, and instance knowledge) of. This type of message is sent from members to leaders of communities, and from leaders to members following a `readFromBuffer` request. For this type of request, the `ontology` slot is set to `KnowledgeClusterOntology.getInstance()`.

### Inform messages

There are five types of `inform` messages that can be sent by `CommunityAgent`'s.

The first type of `inform` message takes a `knowledgeCluster` as content. This represents ontological knowledge that is being shared by an agent. This type of message is sent from members to leaders of communities, and from leaders to members following a `readFromBuffer` request. For this type of inform, the `ontology` slot is set to `KnowledgeClusterOntology.getInstance()`.
The next type of *inform* message takes an abstract content element list `AbsContentElementList` as content. This represents knowledge instances that are being shared by an agent. This type of message is sent from members to leaders of communities, and from leaders to members following a `readFromBuffer` request. For this type of inform, the `ontology` slot is left blank. An agent must use his personal ontology to understand the content, if indeed it is able to.

The next type of *inform* message takes an abstract agent action (`AbsAgentAction`) `updateRole` as content. This agent action takes an integer argument named `newRole`. This represents a command from a leader of a community to a member to update their own view of their role to the new role specified by `newRole`. This type of message is sent from leaders to members following an update of an agent’s role by the leader. It is also sent in reply to a successful `joinCommunity` request message. The reason for sending this type of message is so that the agent is made aware of his change of role within the community. For this type of inform, the `ontology` slot is set to `KnowledgeClusterOntology.getInstance()`.

The next type of *inform* takes an abstract predicate (`AbsPredicate`) `leavingCommunity` as content. This predicate takes an integer argument called `CommunityID`, specifying the community that the sender is leaving. The reason for sending this type of message is so that when an agent leaves a community, then the leader is made aware of the fact, and can remove the list of the members of the community. For this type of inform, the `ontology` slot is set to `KnowledgeClusterOntology.getInstance()`.

The final type of *inform* takes an abstract predicate (`AbsPredicate`) `communityDead` as content. This predicate takes an integer argument called `deadCommunityID`, specifying the community that is dead/dying. The reason for sending this type of message is so that, when a leader decides to destroy a community, it needs to let all members of the community know that this community is dead, so that they stop engaging in communication with the leader about this community. For this type of request, the `ontology` slot is set to `KnowledgeClusterOntology.getInstance()`.

### 4.4.3 Agent Behaviours

The behaviours of an *IndividualAgent* and a *SociableAgent* are implemented in the `IndividualAgentBehaviour` and `SociableAgentBehaviour` classes respectively.

It should also be noted that in the *setup()* method of an *IndividualAgent*, that is called when the agent is created, the user is asked to enter the *goals* of the agent, in the form of concepts that the *IndividualAgent* wants to increase it’s knowledge of.
**IndividualAgentBehaviour**  The IndividualAgentBehaviour is a parallel composite behaviour consisting of four sub-behaviours. The structure of the IndividualAgentBehaviour can be seen in Fig.6. The details of the sub-behaviours follow.

**SelfishRequestBehaviour**  The SelfishRequestBehaviour is a cyclic behaviour. During each cycle the agent performs the following. For each community that the agent is a member of, if there is a request it is wanting to send (in there is a request in his list of requests to send), check it is able to (check that his role allows him to, and that the contribution policy allows him to). If it is, send it and update his count of messages sent and his count of requests sent since it last sent an inform. If the agent is not able to send the request, it tries to do what it is it has to do in order to be able to. This involves checking his list of outstanding requests to answer and trying to answer them. This should ultimately lead to an improvement of role, and count of requests sent since it last sent an inform to zero, so it will be able to send requests again.

To answer a request for knowledge, an agent uses his mapper for the community to check whether or not his personal knowledge overlaps that of the KnowledgeCluster in the request. If it does not, then he cannot answer the request. If it does, he sends any instances he has of concepts and predicates contained in the overlap of his personal knowledge and that of the KnowledgeCluster in the request (if there are any). He also extracts a KnowledgeCluster from his own personal ontology with the same head.
as that of the overlap. If this contains any new information, that was not included in the original request cluster, then he also sends an inform with this KnowledgeCluster.

**ReadMessagesBehaviour** The ReadMessagesBehaviour is a cyclic behaviour. During each cycle the agent receives a message if there is one. It extracts the CommunityAgent.COMM_ID_PARAM from the message to find out which community the message concerns. It then handles the message differently, depending on whether or not it is the leader of the community or not, and whether the message is an inform message, or a request message.

Leaders expect to receive 3 kinds of requests; requests to join a community, a request to read from the buffer, or a requests for knowledge. If the message has any other type, then it is ignored.

If the request is a requests to join a community, the leader decides whether or not to add this agent to the community, assigns it with a role if required, and sends an inform message of type updateRole that lets the agent know he is in the community, and has role specified by the argument newRole.

If the request is a requests to read from the community buffer, the leader first checks that the sender of the message is a member of the community. If it is, the leader extracts the time argument from the message, finds the first message in the buffer after the time specified by time (if there is one), and forwards this message to the agent who made the request. If there is not one, then the leader does nothing.

If the request is a request containing a KnowledgeCluster, the leader just adds the message to buffer, having checked that the sender has correct rights to do so, and also handles the request as a member (see below). This is because leaders are, of course, members of a community as well.

Members only receive one type of request - a request for knowledge in the form of a KnowledgeCluster. Upon receiving such a request they simply add it to their list of requests received, to be handled later (i.e. in the SelfishRequestBehaviour or the SelflessContributorBehaviour.

Leaders expect to receive 3 kinds of informs; an inform of ontological knowledge in the form of a KnowledgeCluster, an inform of instance knowledge in the form of an absContentElementList, or a message from a member agent stating that they are leaving the community.

If the inform message is one of either type of knowledge inform (an inform of ontological knowledge in the form of a KnowledgeCluster, or an inform of instance knowledge in the form of an absContentElementList), then the leader simply the leader adds the message to buffer, having checked that the sender has correct rights to do so. Then the leader checks if the agent deserves a role promotion as a result, and if so, promotes the agent and sends an updateRole inform message to the agent, informing the agent that
its role in the community has changed. Finally, the leader also handles the inform as a member (see below), in order to try to use the knowledge for personal benefit.

If the inform message is a **leavingCommunity** message, the leader simply removes the member who sent the message from the list of members for the community.

Members expect to receive 4 kinds of informs; ontological knowledge in the form of a KnowledgeCluster, instance knowledge in the form of an absContentElementList, an update role message informing the agent new role in the community or a community dead message telling the agent to leave the community.

If the inform message has its **ontology** slot set to nothing, the agent knows that this is an inform of instance knowledge in the form of an absContentElementList. It attempts to use its own personal ontology and mapper to comprehend and add these instances to its own personal knowledge. Of course, this might not be possible, in which case the knowledge is ignored.

Otherwise, if the inform message is an **updateRole** message, the agent extracts the integer argument named **newRole**, and sets its perceived role in the community to this value.

If the inform message is an **communityDead** message, the agent extracts the integer argument named **communityID**, and removes its personal MembersViewOfCommunity for **communityID**.

Finally, if the inform message is an ontological knowledge inform message, in the form of a KnowledgeCluster, the agent adds the cluster to his own personal ontology using his mapper for the appropriate community.

**IndividualMembershipBehaviour** The **IndividualMembershipBehaviour** is a composite behaviour consisting of two sub-behaviours. This behaviour implements an individual agent’s policies on creating, joining, leaving and destroying communities. The details of the sub-behaviours that implement these policies follow.

**SelfishLeaveBehaviour** The **SelfishLeaveBehaviour** is a cyclic behaviour. In each cycle the agent loops through each community it is a member of, checking that the reason (goal) for being in the community still exists in its list of goals. If not, then the agent leaves the community and sends a **leftCommunity** message to the leader of the community. If the agent is the leader of the community, then he destroys the community first. He does this by removing the reference to the community from the community of communities so that no other agents can join the community, sending **communityDead** messages to all members of the community, and then leaving the community itself. Of course, some agents may continue to send this
leader messages for this community until they have read and digested the communityDead message, but these messages will simply be ignored by the agent, which no longer considers itself a member of this community.

**SelfishLeadAndJoinBehaviour**  The SelfishLeadAndJoinBehaviour is a cyclic behaviour. In each cycle the agent loops through it’s outstanding goals and tries to find an existing community which may satisfy that goal by searching the CommunityOfCommunities. If it finds one, it joins it. If it doesn’t find one, it creates a community centered around the goal.

An existing community that may satisfy the agent’s goal is a community whose domain cluster overlaps that of the agent’s goal cluster (the KnowledgeCluster extracted from the agent’s personal ontology, with the goal as it’s head).

In order to join a community, the agent creates an MemberViewOfCommunity with the appropriate CommunityPack and Mapper, and reason for joining set to the goal (for IndividualAgents). The initial role of the agent in the community is the special AgentViewOfCommunity .NO_ROLE role. This represents the fact that the agent is not yet a member of the community. Next, the agent sends a JoinCommunity message to a leader of the community, that it finds from the CommunityPack. Later on (in the ReadMessagesBehaviour), the agent will receive an UpdateRole message, stating that he is a member of the community. Then the agent can update his role, and start interacting with the community as desired.

An individual agent creates a community by extracting it’s goal cluster for a given goal (the KnowledgeCluster extracted from the agent’s personal ontology, with the goal as it’s head) and creating the community around this cluster.

In order to create a community around a given KnowledgeCluster, an agent must select a NormalisedOntology that contains the head concept of the KnowledgeCluster (for an individualistic agent this is the goal itself). Next it creates a new CommunityPack with this goal cluster and NormalisedOntology and adds it to the CommunityOfCommunities. In doing all of this, the agent will have used a Mapper to translate between its personal terms and normalised ones. It therefore creates a MemberViewOfCommunity with the CommunityPack and the Mapper, and sets its role to leader. It also creates a LeadersViewOfCommunity with itself as the initial leader.

**SociableAgentBehaviour**  The SociableAgentBehaviour is a parallel composite behaviour consisting of four sub-behaviours. The structure of the SociableAgentBehaviour can be seen in Fig.7. The ReadMessagesBehaviour and ReadFromBufferBehaviour are detailed above. The details of the other sub-behaviours follow. Note - a SociableAgent never destroys any communities it is the member of. It leaves them in existence for the good of all.
**SelflessContributorBehaviour**  
The SelflessContributorBehaviour is a cyclic behaviour. During each cycle the agent checks his list of outstanding requests to answer and tries to answer them, as detailed under the SelfishRequestBehaviour above.

**SelflessJoinBehaviour**  
The SelflessLeadAndJoinBehaviour is a cyclic behaviour. In each cycle the agent cycles through all of the communities in the community of communities and attempts to join any that it can.

The details for a SociableAgent joining a community are the same as those detailed above for an IndividualAgent in the SelfishLeadAndJoinBehaviour, except that when creating the MembersViewOfCommunity the reason for joining the community is set to AgentViewOfCommunity.ALTRUISM, as there is no goal that constitutes its reason for joining the community. It does it purely for the good of all.

**SelflessLeadBehaviour**  
The SelflessLeadBehaviour is a cyclic behaviour. In each cycle the agent cycles extracts a KnowledgeConcept from it’s PersonalOntology, and creates a community around this concept. The KnowledgeConcept is somewhat arbitrary, but it would make sense to be something the agent may feel the community would benefit from having a community centered around.

The details for a SociableAgent creating a community around a given KnowledgeConcept are the same as those detailed above for an IndividualAgent in the SelfishLeadAndJoinBehaviour.
4.5 Implementation of the System

Details of the Implementation The system that has been implemented has a command-line user interface where all descriptions of the agent activities are written, and where any user input is collected, when decisions need to be made on an agent’s behalf.

Agents are deployed on the jade platform as either an IndividualAgent or a SociableAgent and have some input data files.

The inputs, outputs and decisions collected from the user are exactly as detailed in Sect.4.3. The personal ontology and personal instances of an agent are in are given to the agent in .rdf files, together with any normalised ontologies an agent has. For each normalised ontology, an .xml file detailing any existing mappings between the agent’s personal predicates and concepts and those in the normalised ontology are also given. Any new knowledge that an agent obtains will be written to these files by the agent program.

Differences Between the Design and the Final Implementation

The system has been implemented as in the design, except for a few differences. Firstly, although the system supports multiple policies for communities, which are passed to the community upon creation, the system only currently supports one membership policy, one regulation policy, one role policy and one contribution policy. In a more general system we would add support for many different policies, but due to time constraints the policies have been chosen as detailed below. Also, in the implementation we only have one leader per community. Again, in a more general system we could add support to add more than one leader to a community, but in this first prototype this is not supported.

The policies decided upon are as follows. There is no membership policy, so all types of agents are allowed to join any community. The regulation policy is role based, and the role policy decided upon is one in which agents must contribute a certain minimum number of inform knowledge messages to the community and then they gain a promotion. Agents join a community as a contributor, which means they can only send inform messages to the CommunityBuffer. If they earn a promotion they can become a full member, which means they are also allowed to make requests. The only other roles are the special roles leader and no_role, which is used when an agent has created a MemberViewOfCommunity, but is not yet a member of the community.

Another point that needs making about the implementation is that of the CommunityOfCommunitiesAgent. In the current implementation, the latest version of the CommunityOfCommunities object is sent to and from this agent by the AgentViewOfCommunityOfCommunities object for each agent. There is a synchronisation issue here, and the current implementation contains a bug. This bug will not show itself in situations where the CommunityOfCommunities is not changed very often, i.e. in systems with
only a few agents, especially as every single time the CommunityOfCommunities object is required by an agent, it requests the latest version. But in systems with many concurrent updates to this object the bug could show itself. The bug is as follows, and should be fixed (and would have been done if time had allowed).

Since the latest version of CommunityOfCommunities object is sent to and fro between the CommunityOfCommunitiesAgent and each individual AgentViewOfCommunityOfCommunities, then two individual AgentViewOfCommunityOfCommunities objects could hold different version of the CommunityOfCommunities object, should another agent update it in between the two access of it. If all individual reads and writes to the object were performed by the CommunityOfCommunitiesAgent itself, with updates and reads being requested by the AgentViewOfCommunityOfCommunities using messages, as described in the design, then this bug would be avoided. Unfortunately time constraints have prevented the implementation of the different types of messages that must be created in order to make this possible. The current implementation works for demonstration purposes, and in any case, the CommunityOfCommunitiesAgent is only a convenience to make the system work. In a more general system, we would perhaps extend the abstraction of the concept of knowledge to include knowledge about communities themselves. In this case agents could share this knowledge in the context of communities they themselves create, and there would be no need for a special CommunityOfCommunities.

Finally, the SociableAgent has not been fully implemented due to time constraints, so all agents are IndividualAgents at the current time.

4.6 Using the System

Details on where to get the program, how to install and run the system, as well as details on how to recompile the system in the case of further development can be found in the User Guide in appendix A. This User Guide also details the format of the input .rdf ontology and instance and .xml mappings files and provides some examples.
5 Evaluation

The multi agent based virtual knowledge communities approach is a useful and appropriate method for modelling and implementing distributed knowledge management systems. A prototype system has been developed to show that this approach works for simple examples of knowledge sharing scenarios. Here we evaluate the approach by comparing it to other types of systems that can be used for distributed knowledge sharing.

Figure (8) shows an approach to knowledge sharing with no formal knowledge management system. In this system, all actors rely on themselves alone to identify and locate all knowledge requirements and useful sources for satisfying these requirements. The only interaction between actors is informal. This is clearly not the most useful methodology for maximising knowledge management potential, since it provides no formal means to re-use knowledge found by other actors, nor any means for improving accessibility to data.

Next, let us consider the 'traditional' knowledge management approach (Fig(9)), which is based on standardising all knowledge access through a centralized system. This methodology is great as it allows much knowledge-reuse and accessibility to data from all parts of an organisation; however it is fundamentally flawed in that the knowledge has to be edited in some way for it to be shared in a standard manner. It pays no respect to the context of the information gathered, and therefore not only encourages the sharing of imperfect knowledge, but also discourages the users of a system from using it at all, due to the fact they have to adopt very tight and strong new practices in order to make the system work. This system is too rigid and requires the actors of the system to re-work their knowledge to work in the system, rather than the system doing the work to accommodate the knowledge in its pure form.
Figure 9: Knowledge exchange within a traditional knowledge management system.

Now let us consider the approach presented in this paper. This can be visualised by referring to Fig(10). Here every actor is an autonomous agent, with full control over his own knowledge. The actors do not have to try to fit their own knowledge into another 'view' of the world, rather the mechanics of the system are in place to allow fluid interaction between agents with similar domains of interest, without imposing some kind of fixed idea of the world upon them. This is where things are slightly different to the knowledge nodes approach presented in [2], where the concept of knowledge nodes makes the fixed view of the world more 'local' but still imposes this on to the actors within one knowledge node. The dynamism of this system allows new agents to join the community of communities, and so long as they have one domain concept in common with other agents in the organisation, it is possible for them to find these agents and instigate some kind of exchange of ideas in the context of a community of interest. In essence the abstraction level of an agent and a community has been chosen so as to be as general as possible and to leave the power in the hands of the individuals, rather than in the system itself.

As can be seen in Fig(10), the community approach enables agents to exchange data in a formal manner with other agents on a peer-to-peer basis. As long as agents are in the same community, they have potential to exchange knowledge, and since agents can join any community they wish to, they have potential to exchange knowledge with any agents in the entire organisation in a formal manner, as long as the agents share a domain of interest.
5.1 Example Use Case Showing Strengths of the System

Let us consider a very simple example to demonstrate some of the strengths of the system. Consider an organisation in which one part of the organisation there is an agent responsible for planning holidays. To do so it uses two other agents, that search for hotels and flights in given towns respectively. Then, let us assume in another department of the same organisation, there is an agent performing a very similar task, namely planning holidays, but this one is only using one helper agent, to search for flights.

Figure (11) shows what would happen in this set up with no knowledge management system for result-sharing. Each holiday planner would only consult the agents it knows about and the flight finder agents, who could help each other, would search for themselves and never collaborate with each other.

Next let us consider the solution a traditional knowledge management approach would come up with. A system designer may analyse the entire system and identify that there are two agents that are searching for the
same thing (flights), and two holiday planners. The former can collaborate and pool their findings; the latter is unnecessary replication of a process. Therefore they may design a system similar to that found in Fig (12). Here both departments can use the single plan holiday agent who gets its data from both flight finder agents, and the hotel finder agent.

Next let us consider that in another part of the company a new department is formed in which an agent exists that provides knowledge on which towns are near to other towns. The traditional centralized approach may not be able to locate and use this additional, potentially useful source of knowledge to help in the holiday planning process, especially without the intervention of a system analyst who may need to redesign at least part of the existing system to accommodate this new source of knowledge.

Now let us consider the approach presented in this paper. If we assume the initial organisation presented in Figure (11), we could feasibly expect that one of the agents creates a community centered on the domain of town names. All the existing 5 agents may join this community, as their domains of interest overlap that of ’town name’. The departments containing the plan holiday agents would then continue using the plan holiday agents, without even being aware that they may be accessing knowledge that was provided by an agent in a totally different department.

Now, if we again imagine that a new department is formed in which an agent exists that provides knowledge on which towns are near to other towns. This agent is very likely to join the community concerning town names and should offer some useful information to those agents that are used to plan holidays. Moreover, the department who had an agent that could provide information on which towns are nearby to others may also now be presented with knowledge about which airports and hotels can also be found in these towns. It’s not inconceivable that this information might be of interest to them, although they didn’t build it into their initial system. The resulting can be seen in Figure (13).
Figure 13: The sample organisation with system based on the agent based virtual knowledge communities approach.
6 Conclusions and Future Works

The agent based approach to knowledge management using virtual communities is the appropriate abstraction to use for knowledge sharing in large, heterogeneous, knowledge environments such as corporations and the internet. It can be used to model knowledge management scenarios, and to use the same approach to create useful peer-to-peer (via ‘community’) software systems, which can cope with dynamic changes in the knowledge available from constantly changing different sources.

Since the system is, in it’s most general form, an open one, it requires security policies to ensure only trustworthy agents can access the communities, to prevent malicious attacks from untrustworthy agents. Such policies have not been explicitly built in to the design of the prototype system in this project, but could be implemented by the leaders of the communities on an individual community level in the future. This is vital for a system that wants to be put in to practice.

Additionally, in the realm of corporate knowledge sharing, we cannot assume that the system should be a totally open one, in which any agent can create a virtual knowledge community, as in most (perhaps all) organisations, some kind of organisational hierarchy already exists and must be respected. So, an extension of the general security issue is that the ability and permission to create communities must be related in some way to the existing hierarchy within the organisation the system is being built for.

Another point is that the question of heterogeneity of knowledge sources has not been adequately addressed in this project, and more research needs to be done in to finding solutions for combining different ontologies in order to make progress with this issue. In the prototype system developed, the concept of normalised ontologies was used to avoid this problem, but this was merely a convenience to avoid addressing the problem considered outside the scope of the work. An encouraging idea based upon entropy has been suggested [11] that could direct us towards a potential solution for the problem without using normalised ontologies. This and other ideas need to be researched further.

Simply with regard to the prototype system developed here itself, there is a lot of further work that needs be done if this type of system is to be put in to practice. Firstly, in implementing the general model mentioned in this paper, there are a few things that have not yet been implemented due to time constraints. These were detailed in Sect.4.5, under ”Differences Between the Design and the Final Implementation”. These need to be implemented.

If this implementation with multiple policies was finished, it could be used to find the most appropriate parameters and policies for agents to adopt in a model of a community of virtual knowledge communities. If these can be found, then it could go someway to providing the perfect model on which to build future knowledge management systems. In order to do this,
an implementation with a finer range of agent behaviours also needs to be designed. The prototype system mentioned in this paper only introduced the two extreme forms of agent policies, which we have named the individual approach and the sociable approach.

If the prototype system wants to be used in practice, the user interface would also need to be completely re-designed. The existing user interface is very primitive and was only designed to be functional, in the most minimal sense of the word. A Graphical User Interface should be considered that shows all the details of the communities that an agent is a member of together with his current personal knowledge, including personal ontology, mappings and knowledge instances. This would be a large undertaking in itself, but is a feasible project that could be undertaken in a relatively short period of time.

Finally, in [19], Calmet et. al describe an agent-oriented-abstraction approach to multi agent based systems, such as the one that has been developed in this project. Although the current system leans in the direction of this abstraction mechanism, it does not fully support the approach described in that paper, where essentially an agent is comprised of just knowledge and some decision making system. In order to extend the abstraction in this system, concepts such as the CommunityOfCommunities would have to be removed from being explicitly built in to the system, and instead represented as part of the knowledge of an agent itself, that it can then share with other agents, using a more general concept of a community, that does not rely on the concept of a CommunityOfCommunities. In order to do this, the system would have to be totally re-designed, but the work done in implementing the system in this project at least shows that this would be a worthwhile exercise, if it could be done.
Appendix A - Prototype System User Guide

System Resources

There is a website for this project that can be found at:

www.doc.ic.ac.uk/~mrh00/individualproject.

The website contains the program itself, together with all of the source code of the prototype system, and the example data files.

Installation of the Prototype System

1. Download and install JRE (Java Runtime Environment) from:
   http://java.sun.com/j2se/1.4.2/download.html
2. Download and install JADE from:
3. Download and install the RDFCodec add-on from the JADE 3rd party software and add-ons site:
4. Download and unzip the prototype system itself:
   www.doc.ic.ac.uk/~mrh00/individualproject/prototype/

Set Your Environment Variables (if Using Microsoft Windows XP)

1. Click on Start ⇒ Settings ⇒ Control Panel
2. Select "System"
3. Select the Advanced Tab
4. Click "Environment Variables"
5. Create a new system variable called JAVA_HOME, with the path of your JDK: (if it doesn’t already exist)
   e.g. "C:\Program Files\j2sdk1.4.1"
6. Create another system variable called JADE_PATH, with the path of JADE:
   e.g. "C:\Program Files\jade"
7. Add the paths of the following to the system variable CLASSPATH, seperated by ‘;’
• jade.jar
• jadeTools.jar
• iiop.jar
• base64.jar
• RDFCodec.jar
• virtualKnowledgeCommunities.jar

e.g. "%JADE_PATH%\lib\jade.jar; %JADE_PATH%\lib\jadeTools.jar;
%JADE_PATH%\lib\iiop.jar; %JADE_PATH%\lib\base64.jar; %JADE_PATH%\addons\RDFCodec2\lib\RDFCodec.jar;
d:\prototypeVirtualKnowledgeCommunities\lib\virtualKnowledgeCommunities.jar"

Running the Virtual Knowledge Communities Platform
Once you have set these environment variables you should be able to run the
prototype system (which runs JADE and deploys the CommunityOfCommunitiesAgent)
by simply running run_communityOfCommunities.bat from the directory
where you unzipped prototypeVirtualKnowledgeCommunities.zip.

Setting up the Virtual Agent Community Agents
The next step is to set up the agents you wish to deploy onto the jade plat-
form, and into the virtual knowledge community of communities.

Each agent needs to be given a name (which we will call here ⟨agentname⟩),
and then the following files need to be created. Empty templates for all the
files that an agent requires can be found in the prototype .zip file, under the
following directory:

agent_data_files\template

• ⟨agentname⟩_personal_ontology.rdf
  This file contains the personal ontology of the agent, stored in .rdf
  format. This .rdf represents a KnowledgeCluster. The .rdf format
  is described hereafter.

A blank personal_ontology looks as follows.

<?xml version="1.0" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#"
  xmlns:fipa-rdf="http://www.fipa.org/schemas/FIPA-RDF#">
  <rdf:object>
    <fipa-rdf:CONTENT_ELEMENT>
In between the `<knowledgeConcepts>` tags, we describe the `knowledgeConcepts` that the agent’s personal ontology contains by listing each one as follows, and putting the concept name in place of `PERSONAL_ONTOLOGY_CONCEPT_NAME`.

```xml
<rdf:li>
  <rdf:object>
    <fipa-rdf:type>knowledgeConcept</fipa-rdf:type>
    <name>PERSONAL_ONTOLOGY_CONCEPT_NAME</name>
    <slots>
      </slots>
  </rdf:object>
</rdf:li>
```

In between the `<slots>` tags, we describe the `ConceptSlots` of the `knowledgeConcepts`, which represent attributes of a concept, by listing each one as follows.

```xml
<rdf:li>
  <rdf:object>
    <fipa-rdf:type>conceptSlot</fipa-rdf:type>
    <optionality>OPTIONALITY</optionality>
    <slotMin>SLOTMIN</slotMin>
    <aggType>AGG</aggType>
    <slotName>CONCEPT SLOT_NAME</slotName>
    <slotType>CONCEPT_TYPE</slotType>
    <slotMax>SLOTMAX</slotMax>
  </rdf:object>
</rdf:li>
```
The OPTIONALITY slot is either -1, 0 or 1. 1 means that the slot is optional, 0 means that it is compulsory. -1 means that it is unspecified. The AGG slot specifies the aggregation type of the slot. If there is no aggregation, i.e. if the slot represents a single-valued attribute, then it takes the value none. If the slot does represent an aggregation, the SLOTMIN and SLOTMAX values specify the minimum and maximum number of elements respectively. Finally, the CONCEPT SLOT_NAME is the name of the attribute that the slot represents, and the CONCEPT SLOT_TYPE is the type of object that this is. The type can be the name of another knowledgeConcept specified in the ontology file, or it can be one of the basic types provided by the BasicOntology class in JADE, e.g. BOString.

The knowledgePredicates in the personal ontology are described in a similar way. In between the <knowledgePredicates><rdf:Seq> </rdf:Seq></knowledgePredicates> tags, we describe the knowledgePredicates that the agent’s personal ontology contains by listing each one as follows, and putting the concept name in place of PERSONAL_ONTOLOGY_PREDICATE_NAME.

<rdf:li>
<rdf:object>
<fipa-rdf:type>knowledgePredicate</fipa-rdf:type>
<name>PERSONAL_ONTOLOGY_PREDICATE_NAME</name>
<arguments>
<rdf:Seq>
</rdf:Seq>
</arguments>
</rdf:object>
</rdf:li>

In between the <arguments><rdf:Seq> </rdf:Seq></arguments> tags, we describe the ConceptArguments of the knowledgePredicates, by listing each one as follows.

<rdf:li>
<rdf:object>
<fipa-rdf:type>conceptArgument</fipa-rdf:type>
<optionality>OPTIONALITY</optionality>
<argName>ARGNAME</argName>
<aggType>AGG</aggType>
<argType>ARGTYPE</argType>
<argMax>ARGMAX</argMax>
<argMin>ARGMIN</argMin>
</rdf:object>
</rdf:li>
The slots in this definition have take the same values as those described for ConceptSlots.

- (agentname)_personal_instances.rdf

This file contains the personal instances of the agent, stored in .rdf format. This .rdf represents an AbsContentElementList of instances. The .rdf format is described hereafter.

A empty set of personal instances looks as follows.

```xml
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#"
xmlns:fipa-rdf="http://www.fipa.org/schemas/FIPA-RDF#">
  <rdf:object>
    <fipa-rdf:CONTENT_ELEMENT_LIST>
      <rdf:Description>
        <fipa-rdf:type>PREDICATE_NAME</fipa-rdf:type>
      </rdf:Description>
    </fipa-rdf:CONTENT_ELEMENT_LIST>
  </rdf:object>
</rdf:RDF>
```

In between the `<fipa-rdf:CONTENT_ELEMENT_LIST>` tags, we describe the personal instances that the agent has contains by listing each one as follows, and putting the name of the predicate that this is an instance of in place of `PREDICATE_NAME`.

```xml
<fipa-rdf:CONTENT_ELEMENT>
  <rdf:Description>
    <fipa-rdf:type>PREDICATE_NAME</fipa-rdf:type>
  </rdf:Description>
</fipa-rdf:CONTENT_ELEMENT>
```

For each argument that the predicate with name `PREDICATE_NAME` takes, we fill in the information of the instances as follows (after the `<fipa-rdf:type>PREDICATE_NAME</fipa-rdf:type>` tag). If the argument is of simple type, e.g. BOString, then the argument instance is described like this:

```xml
<PREDICATE_ARG_NAME>
  <rdf:Description>
    <PREDICATE_ARG_SLOT_NAME>ARGUMENT_VALUE</PREDICATE_ARG_SLOT_NAME>
  </rdf:Description>
</PREDICATE_ARG_NAME>
```
</PREDICATE_ARG_SLOT_NAME>
</rdf:Description>
</PREDICATE_ARG_NAME>

Where PREDICATE_ARG_NAME is the name of the argument, and PREDICATE_ARG_SLOT_NAME is the name of a parameter for this type of argument. The value of this parameter is specified by ARGUMENT_VALUE.

Otherwise the argument instances is described like this:

</PREDICATE_ARG_NAME>
<rdf:Description>
<fipa-rdf:type>PREDICATE_ARG_TYPE</fipa-rdf:type>
<PREDICATE_ARG_SLOT_NAME>
PREDICATE_ARG_SLOT_DESC
</PREDICATE_ARG_SLOT_NAME>
</rdf:Description>
</PREDICATE_ARG_NAME>

Where PREDICATE_ARG_TYPE specifies the more complex type of the argument, and PREDICATE_ARG_SLOT_DESC is a description of the instance of a slot of the concept argument, which takes either a string value (in the case of no aggregation) or a sequence of the following form when the slot is an aggregation of instances:

<rdf:Seq>
<rdf:li>
<rdf:object>
<fipa-rdf:type>CONCEPT_SLOT_TYPE</fipa-rdf:type>
<CONCEPT_SLOT_SLOT_TYPE>
CONCEPT_SLOT_SLOT_VALUE
</CONCEPT_SLOT_SLOT_TYPE>
</rdf:object>
</rdf:li>
</rdf:Seq>

Where CONCEPT_SLOT_TYPE is the type of concept the slot takes, CONCEPT_SLOT_SLOT_TYPE is the type of concept of one of the slots of the concept slot, and CONCEPT_SLOT_SLOT_VALUE is the value of the instance for this slot.

• RDFCodec.properties

This is just a blank file that is required to prevent the RDFCodec throwing a FileNotFoundException.

The following files are also required for each normalised ontology that an agent knows about.
This is an .rdf file that represents a normalised ontology in the form of a KnowledgeCluster. It takes an identical format to that of a personalised ontology, which was described above.

This is an .xml file that represents the mappings an agent has between its personal concepts and predicates, and those in the normalised ontology. It takes the following format:

```xml
<?xml version="1.0"?>
<mappings>
  <mapping personalTerm='⟨personalTermName⟩' normalisedTerm='⟨normalisedTermName⟩' />
</mappings>
```

Where each `<mapping>` tag represents one two-way mapping between a personal predicate or concept with the name `personalTermName`, and a normalised predicate or concept with the name `normalisedTermName`, that can be found in the normalised ontology given by the `<normOntName>` in the filename.

Examples of agent’s instances, personal ontology’s, normalised ontologies and mappings can be found under:

`agent_data_files`

Running the Virtual Agent Community Agents

To run the virtual agents, ensure that JADE is running with the community of communities agent (as detailed above), and then just type the following at the command prompt:

```
java -classpath %CLASSPATH% jade.Boot -container agentName:agent.IndividualAgent
```

(or double click the .bat file associated with this agent).

Example Agents

Two agents have been created to demonstrate the program. They are called `mark` and `jose`. They can be found here:

```
agent_data_files\mark
```

and

```
agent_data_files\jose
```
Mark and Jose both have the same normalised ontology, called PHYSICALOBJECT, but have different personal names for all the concepts in the ontology. Jose already has the mappings from his personal concepts to the ones in the normalised ontology contained in his mapping file. Mark does not. Moreover, Jose has knowledge about the predicate owns, whereas Mark does not. Jose has instances of this predicate also.

If we perform the following test run, we will see how knowledge sharing can take place, and mark can gain both onotological knowledge about the concept owns, as well as instances, even though this concept is not included in the normalised ontology, and mark and jose have different names for all the terms they share in common.

**Details of the test run**

Run the community of communities on JADE by double clicking on run_communityOfCommunities.bat.

Before deploying mark and jose, you may want to backup the original data files for mark and jose to see how they change after completing the test run, and so they can be used again. Also, analyse the data files before the run to see how they change later.

To run the agents simply double click on run_mark.bat and run_jose.bat respectively.

In the mark window:

1. Set mark’s goal to ‘mark_ITEM’. Then type ‘done’.
2. When mark tries to create a community around 'mark_ITEM', type the name of the normalised ontology 'PHYSICALOBJECT'.
3. Map more appropriate names to the concepts mark_TRACK, mark_CD, mark_BOOK for use in the community, for example TRACK, CD, BOOK. Map the personal concept mark_ITEM to the normalised term normalised_THING, as these concepts are intended to be the same.

At this point, mark creates the community and adds a request for knowledge on mark_ITEM (normalised_THING) to the community’s buffer. Now, in the Jose window:

1. Set jose’s goal to ‘jose_PHYSICALOBJECT’. Then type ‘done’.
2. Jose should try to join the community that mark created, as he finds that the domain of this community overlaps with his goal. When jose tries to join the community, map the concepts jose_TRACK, jose_CD,
jose_BOOK to TRACK, CD, BOOK respectively. You can also map a more appropriate name to the predicates jose_OWNS for use in the community, for example OWN.

At this point, Jose sends a join request to mark to join the community. We can see that mark responds to this by looking in the mark window.

3. Jose will now also try to create a community with domain jose_PHYSICALOBJECT, as one does not exist. Enter the name of the normalised ontology ‘PHYSICALOBJECT’ to use for this community. Note that at this point, Jose imports his normalised mappings from file.

4. Map more appropriate names to the concepts jose_TRACK, jose_CD, jose_BOOK and predicate jose_OWNS for use in the community, for example CD_TRACK, COMPACT_DISK, BOOK and OWNS_ITEM.

At this point, Jose creates the community and adds a request for knowledge on jose_PHYSICALOBJECT (normalised_PHYSICALOBJECT) to the community’s buffer. Jose also receives the update role message to say he has been accepted in to mark’s community, and reads mark’s request from the community buffer. Since Jose is an individual agent, he wants to achieve his goals, but is unable to make requests in mark’s community as of yet, due to the contribution and role policies. So, since he can, he sends clusters and instances in response to mark’s request.

Meanwhile, in the mark window, mark has discovered Jose’s new community:

1. Map the concepts mark_TRACK, mark_CD, mark_BOOK to CD_TRACK, COMPACT_DISK, BOOK respectively, for use in this community.

At this point, Mark joins sends a join request to the community, which Jose responds to by adding Mark to the community and sending an update role message. After this, mark receives the inform of knowledge in community 1. Since the received ontological knowledge overlaps his personal ontology, he can use the knowledge. He adds the new concepts, but not ones he already has.

2. Although Mark obtains some new knowledge in doing this (we can see from the file mark_personal_ontology.rdf, that mark now understands the predicate OWNS), we say he has not yet satisfied his goals. Type ‘n’ in response to the question concerning this.
3. Now Mark receives the second inform message that Jose sent, this time containing instances. In order to accept this we have to create mappings between mark\_PHYSICALOBJECT, mark\_HORSE and normalised\_PHYSICALOBJECT, normalised\_HORSE respectively.

He adds the mappings to his list of normalised mappings, and the new instances to his personal instances (check the files mark\_mappings\_for\_normalised\_ontology\_PHYSICALOBJECT.xml and mark\_personal\_instances.rdf).

4. Let us say that Mark has now achieved his goal. Type 'y' in response to the question concerning this. He now leaves both communities, destroying the community he is the leader of in the process. We can see that he communicates with Jose in order to do this.

**Changing and Re-Compiling the Code**

**APACHE ANT**

Apache ANT can be used to build all the class files within the project. It can be downloaded from here:

http://ant.apache.org/

The build.xml file for compiling the project can be found in the prototypeVirtualKnowledgeCommunities zip file/folder. To use ant to build the project simply type:

    ant virtualKnowledgeCommunities

**JBUILDER**

JBuilder can also be used as a development environment to build and compile agents.

http://www.borland.com/jbuilder/

Details on how to set up JBuilder to use JADE can be found in the JADE documentation in the article:

"Administering the JADE platform. Tutorial for beginners" by David Grimshaw - Ryerson University

Available from:

Bibliography


[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. ISBN 0-201-63361-2