Applying Conflict Management Strategies in BDI Agents for Resource Management in Computational Grids[*]

Omer F. Rana[1,2] **Michael Winikoff**[1], **Lin Padgham**[1], **James Harland**[1]

[1]School of Computer Science and Information Technology,
RMIT University, Melbourne, Australia
Email: {winikoff,linpa,jah}@cs.rmit.edu.au
[2]Department of Computer Science, Cardiff University, UK
Email: o.f.rana@cs.cf.ac.uk

Abstract

Managing resources in large scale distributed systems – "Computational Grids", is a complex and time sensitive process. The computational resources being shared vary in type and complexity, and resource properties can change over time. An approach based on interacting software agents is presented, where each resource manager and resource requester is modelled as a BDI (Belief-Desire-Intention) agent. The proposed approach can help resolve conflicts that arise during resource discovery and application scheduling, and enables site autonomy to be maintained. The modelling and detection of conflicts is important in the context of this work, to enable each resource and application to respond to changes in the environment. We propose a BDI based framework that can be used to model agents that represent resources and applications – and outline properties that each must maintain.

1 Introduction

An intelligent agent is able to make rational decisions, i.e., blending proactiveness and reactiveness, showing rational commitment to decisions made, and exhibiting flexibility in the face of an uncertain and changing environment. Agents offer new ways of abstraction, decomposition, and organisation that fit well with our natural view of the world and agent oriented programming is often considered a natural successor to object oriented programming [Jennings, 2001]. It has the potential to change the way we design, visualise, and build software in that agents can naturally model "actors" – real world entities that can show autonomy and proactiveness. Additionally, social agents naturally model (human) organisations ranging from business structure & processes to military command structures. A number of significant applications utilising agent technology [Jennings and Wooldridge, 1998a] have already been developed, many of which are decidedly non-trivial, such as the military simulation work undertaken with dMars containing thousands of plans [Tidhar et al., 1998].

In this paper we apply intelligent software agents, using the Belief-Desire-Intention (BDI) model [Wooldridge, 2000, Georgeff and Rao, 1998, Rao and Georgeff, 1992] (more details about BDI agents can be found in section 3), to the problem of resource management in distributed systems. The resource management problem in distributed systems (in its simplest form) consists of, (1) selecting a set of resources on which to execute tasks generated from an application, (2) mapping tasks to computational resources, (3) feeding data to these computations, and (4) ensuring that task and data dependencies between executing tasks are maintained. Generally, relations between tasks are defined using a task graph – which provides a partial ordering on task execution. Each node within such a graph represents a computation, and arcs represent data or control relationships. Once a task graph has been specified, the next stage involves resource selection or discovery, to identify suitable computational engines from a pool, typically homogeneous, based on criteria ranging from licensing constraints, processor(s) capability(ies), execution costs, and background workload. A good overview of such cluster management systems can be found in [Baker, Fox and Yau, 1996]. In [Rana et al., 2001] an agent based MatchMaking service is described, which acts as a "yellow pages" service to discover resources of interest. This study assumes that resources can be heterogeneous, and their properties can change over time. A match between task and resource properties is achieved by finding commonalities (either syntactic, contextual or semantic) between task and resource properties. An implementation based on JKQML [JKQML, 1999] is also provided in the study to demonstrate the concepts. The agents however undertake simple activities, and do not have associated behaviours that can be used to adapt their operations over time.

We extend the work in [Rana et al., 2001] to include rational agents based on the BDI model. A rational agent executes a plan (from a pre-defined plan library) to achieve local goals, and can retry alternate plans if a goal cannot be achieved. To utilise this model, it is necessary to translate the resource management problem into goals that need to be satisfied locally within each agent, based on the role that an agent undertakes within the system. We identify three roles that are necessary for such a resource management system, (1) a resource agent role, (2) an application agent role, and (3) a middle (broker) agent role. These are described in greater detail in section 2. Expressing the resource management problem in this way supports a de-centralised management strategy, whereby each agent in the system is responsible for managing its local goals, and is particularly useful in an environment (such as Computational Grids [Foster and Kesselman, 1999]) where resource capabilities and application demands can vary significantly over time.

2 Resource management

A three tier system is considered, based on BDI Application Agents (AAs), Broker Agents (BAs) and BDI Resource Agents (RAs) as illustrated in figure 1. Each AA is responsible for managing the execution of a program described as a task graph. An AA can manage multiple programs (task graphs) and undertakes a similarity check between the task graphs it manages to identify common tasks. A task graph is an ordered set, consisting of tasks θ and arcs η,

$$Task\,Graph(TG) = (\theta, \eta)$$

[*]We would to thank Agentis International and the ARC (under grant C49906798, "*Real-time reasoning about interacting activities in agent systems*") for their contribution to this work.

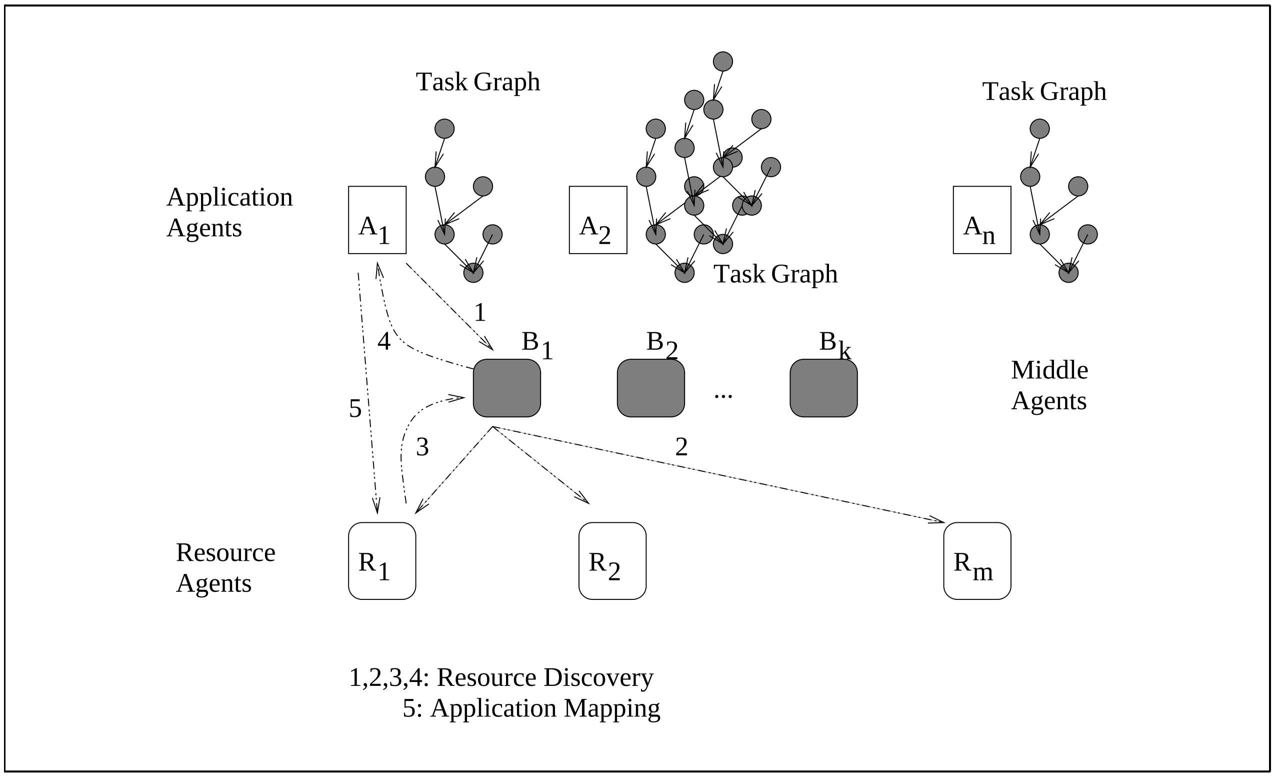

Figure 1: *Overall system architecture. The figure contains three kinds of agents: Application Agents (A), Broker or Middle Agents (B) and Resource Agents (R). Application agents are responsible for handling task graphs generated from computational applications. Broker agents support the application agents in locating resources. Each resource agent is responsible for managing a particular computational resource*

$$\eta = \eta_{in} \cup \eta_{out}$$

where each θ is an executable unit. The set of arcs η is the union of input and output arcs, and each arc carries a label, a type, and data. Two tasks θ_i and θ_j are said to be *identical*[1] if:

$$(\eta^i_{in} \cap \eta^j_{in} \neq \emptyset) \wedge (\eta^i_{out} \cap \eta^j_{out} \neq \emptyset)$$

and two tasks are said to be *similar* if:

$$(\eta^i_{in} \cap \eta^j_{in} \neq 0)$$

Activities undertaken by each of the different types of agents illustrated in figure 1 are described in the following sections.

2.1 Application Agents

An AA is responsible for managing one or more task graphs. As identified previously, each task graph is a directed graph, with nodes representing executable tasks, and links representing dependencies. For tasks that can be concurrently executed, an AA must decide an ordering – especially if tasks belonging to different graphs can be shared. In cases where common tasks can be detected, such tasks are given a higher precedence (execution priority). Each AA must therefore perform a static analysis of a task graph to determine whether tasks can be shared across graphs.

Each AA maintains a plan library (see section 3) which can be used to change the ordering of tasks, the discovery of common tasks, the decomposition of tasks, and the aggregation (grouping) of tasks. Each plan is a well defined logic formula, which is triggered based on the current state of the AA, and its priorities.

[1]The tasks are not identical in the conventional sense of being indistinguishable, merely in the (weaker) sense of having identical inputs. A more detailed definition of 'similar' and 'identical' is provided later for specifying BDI behaviours (see section 3.1).

Let $\Re$ be a set of resources, and $\Im$ be a set of tasks, where

$$\Re(t) = \{R_1(t), ..., R_n(t)\}, \Im = \{T_1, ..., T_m\}$$

each T_i can be obtained from a task graph, or may be specified directly by the application developer. Every $R_j(t)$ specifies the state of resource 'j' at time 't'. Associated with each T_i is an execution time, such that the total execution time of an application with 'k' tasks (T_{App}) can be specified as:

$$T_{App} = \sum_{i=1}^{k} T_i$$

Each T_i consists of the time to acquire data for a task, perform the execution, and write results to a file system. If tasks are run in parallel, the total execution time is the sum of the maximum task execution times (at each stage of execution) of the parallel tasks (assuming that tasks run concurrently). The objective of the AA is to find a mapping function $\Phi : \Im \rightarrow (j,t), 1 \leq j \leq n$, which maximises the utilisation of each resource, and minimises T_{app}. The function Φ allocates tasks to resources; $\Phi(T_i) = (j,t)$ represents the presence of task T_i on resource R_j at time t.

We consider 't' to be discretised, and assume that the total execution time of an application can be specified in units of this parameter. From this definition, a resource management system would monitor the state of the system (i.e. the properties of T_i and R_j) and aim to find a function Φ such that the execution time of an application for each AA is minimised, and the utilisation for each RA is maximised. Traditionally the determination of Φ has been considered as an optimisation problem, over tasks and resources. This problem has been addressed in various ways, generally involving two simplifying assumptions,

(1) ‘n’ is fixed and all members of $\Re$ are identical, (2) ‘m’ is fixed, and all members of $\Im$ are pre-defined (suggesting a static schedule). Additionally, the mapping Φ is generally performed by a centralised scheduler (allocator), with some support for dynamic task creation (i.e. variable ‘m’) provided via an additional dispatcher. In this latter case, the dispatcher works with a resource information service to generate $\Re(t)$ at fixed time intervals, or whenever a new T_i needs to be allocated to a resource. In order to overcome these restrictions, we propose a de-centralised mechanism, which can cater for variable ‘n’ and ‘m’.

2.2 Resource Agents

A RA is responsible for managing access to services being offered on a given resource. The RA also monitors the state of the resource, and makes these parameters available to Broker and Application agents. Each RA maintains a plan library (similar to an AA) based on the type of resource it manages. For computational resources this can include scheduling operations on the resource, re-ordering a given schedule, and pre-empting executing tasks. Based on its current state (beliefs), the RA makes one or more plans active, and executes these in order to achieve its goal of improving resource utilisation.

Consider an application composed of one AA and two RAs. The application task graph consists of three tasks, where the granularity of each task may range from a sub-routine, to a complete application. We consider each resource to specialise in a particular operation, but be capable of supporting all three task executions. Based on our task graph representation, we could define[2]: $RA_1(\theta_1) = 10, RA_2(\theta_1) = 12, RA_1(\theta_2) = 8, RA_2(\theta_2) = 4, RA_1(\theta_3) = 30, RA_2(\theta_3) = 32$. This is summarised in the following table:

	θ_1	θ_2	θ_3
R_1	10	8	30
R_2	12	4	32

Table 1: *Execution times on Resource Agents*

Hence, the total execution time on RA_1 would be 48 units, and on RA_2 would also be 48 units. However, if θ_1 was run on RA_1, θ_2 on RA_2, and θ_3 on RA_1 then the total execution time would be 44 units (assuming that task assignment is static). In the second scenario, RA_2 may specialise in performing tasks of type θ_2, and should be identified as the most suitable resource by the broker. It is in the interest of the AA to map task θ_2 to resource RA_2 to minimise its execution time. If we consider agent AA_2, consisting only of tasks of type θ_2 from AA_1, then this agent will find a match for all of its tasks from resource RA_2, and it must now find a mechanism to execute all its tasks on this resource. This leads to a conflict between AA_1 and AA_2 in determining who should access RA_2. This conflict may be overcome by other parameters, such as the priority associated with the application being managed by AA_1 over AA_2, and whether some tasks being managed by the two AAs are similar in some way. It is also possible for some resources to be incapable of executing certain tasks – in which case the conflicts are much easier to resolve. Since none of the application or resource agents are aware of the complete state of the other agents at any one time, it is hard for them to optimise their schedule for the complete application. This holds in the context of Computational Grids in particular, where a resource cannot influence task execution at other resources, or determine the selection of tasks from particular applications.

Handling such conflicts efficiently is essential in making more effective use of resources. In some situations, it may be possible to overcome some of these conflicts, whilst in others, it may only be possible to flag the existence of such conflicts.

[2]Where $RA_j(\theta_i)$ is the execution time T_i on resource R_j.

2.3 Broker Agents

The BAs can undertake different roles within such a system, offering services such as a certificate granting service, a matchmaking service etc. Hence, a BA may restrict interaction with AAs and RAs based on access criteria, such as restricting access to RAs from one or more administrative domains, access to tasks from RAs based on the types of resources currently available, and based on periods of access for particular types of resources. Each RA is responsible for advertising the capability of a resource to a BA, and also for monitoring activities undertaken on the resource. For instance, an RA for a computational resource would monitor the number of processes currently active on the resource, the usage of local memory etc. These metrics are then reported to the AA managing a program on a given resource, and to a BA which requests this information to determine suitable resources to which tasks may be mapped. The data model for exchanging this information is provided in [Rana et al., 2001].

3 BDI Agents

The BDI model [Georgeff and Rao, 1998, Rao and Georgeff, 1992] is a popular model for intelligent agents. It has its basis in philosophy [Bratman, 1987] and offers a *logical theory* which defines the mental attitudes of Belief, Desire, and Intention using a modal logic; a *system architecture*; a *number of implementations of this architecture* (e.g. PRS [PRS, 2001], JAM [IRS, 2001], JACK [JACK, 2001]); and *applications* demonstrating the viability of the model.

The central concepts in the BDI model are [Georgeff and Rao, 1998, page 144]:

Beliefs: Information about the environment; *informative*.

Desires: Objectives to be accomplished, possibly with each objective’s associated priority/payoff; *motivational*.

Intentions: The currently chosen course of action; *deliberative*.

Plans: Means of achieving certain future world states. Intuitively, plans are an abstract specification of both the means for achieving certain desires and the options available to the agent. Each plan has (i) a body describing the primitive actions or sub-goals that have to be achieved for plan execution to be successful; (ii) an invocation condition which specifies the triggering event, and (iii) a context condition which specifies the situation in which the plan is applicable.

We shall use the notation and execution model of AgentSpeak(L) [Rao, 1996] as an exemplar of BDI systems. An AgentSpeak(L) agent consists of a belief set, and a collection of plan clauses. Each plan clause is of the form

$$\textit{goal} : B_1 \wedge \ldots \wedge B_n \leftarrow S_1; \ldots ; S_m$$

where each B_i is a belief, and each S_i is either an action (a), or a subgoal (α_{sub}).

The execution model of AgentSpeak consists of the following steps:

1. The agent selects an event e (note that goals are an event type)
2. The agent generates all plans with matching invocation conditions
3. From these relevant plans the agent identifies those with satisfied preconditions

4. If there are several plans, one is chosen nondeterministically

The plan is then added to the intention stack. The intention stack is executed by popping the topmost plan of an intention and performing the first (unperformed) S_i. If S_i is an event, then it is posted, and if it is an action, then it is executed.

In the interests of conciseness we shall use π to denote an agent's plan set, π_i to denote the i^{th} plan clause, α to denote the goal which triggers π_i, and X to denote $B_1 \wedge \ldots \wedge B_n$. Additionally, we use $\pi_i(X, \alpha)$ to denote the body of π_i. Assuming that a plan consists of a number of actions a, we can define:

$$\pi_i(X, \alpha) = (a_1(p), a_2(p), ..., a_k(p))$$
$$head(\pi_i(X, \alpha)) = (a_1(p)),$$
$$tail(\pi_i(X, \alpha)) = (a_2(p), ..., a_k(p))$$

where each action $a_i(p)$ corresponds to a well defined operation that an AA can perform on a task graph, or an RA can perform on its local schedule (or executing tasks). Each action within a plan results in an update on the properties (p) of a task or a resource. A plan is therefore not an atomic operation, and it is possible for a plan to be interrupted during execution. An agent can measure its environment after running each action within the plan, and based on this determine if plan execution should continue.

3.1 Specifying BDI behaviours

Each RA and AA is aiming to maximise its utilisation and application execution time, respectively. For AAs that manage multiple applications, the maximisation is over multiple application task graphs. AAs compete for resources in order to minimise the execution time for a task graph that they manage. Similarly, each RA is aiming to maximise its utilisation over all available applications. Resource agents compete for tasks from AAs, based on the particular capabilities of the RA.

Typically, the actions performed by the RAs correspond to well defined operations on the agent's local schedule (or executing task). The actions performed by AAs correspond to well defined operations on a task graph. Each action influences the properties of a task graph or a resource, and can be defined as a function which modifies the properties of a resource in some way (discussed later). The time to execute a plan T_{π_i} can be expressed as:

$$T_{\pi_i} = t_{deliberate} + t_{execute(\pi_i)}$$

The value of $t_{deliberate}$ depends on the complexity of the agent, the complexity of the environment, and on the size of the plan library. $t_{deliberate}$ specifies the time it takes the agent to select a plan – when multiple plans may be selection for a given event/belief. This time is applicable to all BDI agents – AA, BA and RA here. We assume that $t_{execute(\pi)} \gg t_{deliberate}$, and therefore, we can approximate $t_{deliberate} + t_{execute(\pi)}$ with $t_{execute(\pi)}$. The validity of this assumption depends on the criteria mentioned above. Each AA attempts to minimise execution time for a task graph, hence the goal for an AA is:

$$Goal(agent(AA)) = \text{minimise} \sum_{i=1}^{n} t_i(p)$$

where $t_i(p) = t_{execute(\pi)} + t_{execute(i)}$ – i.e. execution time is the sum of plan execution and the time to execute a task from the task graph. In an AA, the properties 'p' refer to the current ordering of tasks, the change in dependencies between the tasks etc. Hence, running an action $a_i(p)$ within a plan π_i will result in a change of these properties. A plan may also be represented as a task graph, in which case the AA must determine whether the plan should also be run remotely. We assume that the machine hosting the AA will be able to execute the plan locally. Similarly, the goal function for a RA can be defined as:

$$Goal(agent(RA)) = \text{maximise} \sum_{j=1}^{m} C(p_j)$$

where $C(p_j)$ represents the capacity of the resource on a given property p_j – where resource properties can range from local memory, CPU utilisation etc. Alternatively, an RA may attempt to maximise utilisation on one or more individual properties. If there are **P** properties in total, then the goal function is:

$$Goal(agent(RA)) = \bigwedge_{p \in \mathbf{P}} \text{maximise } C(p)$$

where *maximise*$(C(p))$ represents the goal of maximising the achieved value for a given resource property. For instance, a resource may aim to maximise its memory usage, but not CPU utilisation. By selective maximisation of a particular property, an RA can aim to achieve a particular behaviour over a given time period. The beliefs (X) of an AA and RA correspond to the value of these properties that are measured at any time.

Based on the plans that each AA and RA maintains, it is possible for multiple plans to be runnable at a given time, if pre-conditions to such plans match. In this case, the agent needs to choose between the available plans (in reality, this may be random, or based on particular administrative policy). Plan ordering is left to the AA or RA, and it is possible for AA or RA agents to have the same plan library, but different plan ordering. Each AA agent maintains the following beliefs:

$task(\theta_i, In, Out, f)$**:** which says that the task labelled θ_i has input links *In* $= \langle L_i, L_j, \ldots \rangle$, output links *Out* $= \langle L_k, L_l, \ldots \rangle$ and function f which takes values from each input link and produces a tuple of values, one for each output link.

$link(L_i, T, V)$**:** which says that the link labelled L_i has type T and value V which can either be a value of type T or the distinguished value $\perp$ indicating that no value has yet been determined for the link.

For example, a simple mathematical task which adds its three inputs might be specified as the beliefs:

task(θ_1, $\langle L_1, L_2, L_3 \rangle$, $\langle L_4 \rangle$,
$f = \lambda \langle x, y, z \rangle \mapsto \langle x + y + z \rangle$)
link(L_1, $\mathbb{N}$, 3)
link(L_2, $\mathbb{N}$, 6)
link(L_3, $\mathbb{N}$, 3)
link(L_4, $\mathbb{N}$, $\perp$)

Generally the exact operation performed by function f is not known, and it often corresponds to an executable (binary) program. In section 2 we have provided a general definition of what is necessary for two tasks to be *similar* and *identical*. To make this definition more specific, we relate these ideas to the beliefs of each agent. Hence, a task θ_i is *runnable* if all of its input links have supplied values: *runnable*$(\theta_i) \Leftrightarrow task(\theta_i, In, Out, f) \wedge \forall L_i \in In \,.\, link(L_i, T, V) \wedge V \neq \perp$

Two tasks are similar of they have common inputs, that is the same number of input links, and a correspondence between the two sets of input links, where two inputs are considered the same if they have the same value. For convenience we define the notation In^* to be the *multi-set* of types and values corresponding to the labels in In, so for the example above $In^* = \langle L_1, L_2, L_3 \rangle^* = \{(\mathbb{N}, 3), (\mathbb{N}, 6), (\mathbb{N}, 3)\}$.

$$In^* \equiv \{(T_i, V_i) | L_i \in In \wedge link(L_i, T_i, V_i)\}$$

We can now define similarity: *similar*$(\theta_i, \theta_j) \Leftrightarrow task(\theta_i, In_i, Out_i, f_i) \wedge task(\theta_j, In_j, Out_j, f_j) \wedge {In_i}^* = {In_j}^*$

We also have a notion of "identical" tasks. Two tasks are identical if they have the same number of inputs, the same number of outputs, and the same values and types for inputs. *identical*$(\theta_i, \theta_j) \Leftrightarrow task(\theta_i, In_i, Out_i, f_i) \wedge task(\theta_j, In_j, Out_j, f_j) \wedge {In_i}^* = {In_j}^* \wedge \#Out_i = \#Out_j$ Note that two identical tasks do *not* necessarily compute the same functions – they merely have the same inputs, and can be conveniently allocated together.

We assume that we have an algorithm to identify similar and identical tasks in the AA's task graph(s). We treat this as a capability of the agent and assume that it examines the agent's beliefs and adds beliefs of the form *similar*$(\{\theta_i, \theta_j, \ldots\})$ and *identical*$(\{\theta_i, \theta_j, \ldots\})$. An application agent uses these beliefs (*identical*, *similar*, *task*, and *link*) in the execution of its plans.

A task is allocated when an AA sends a message to an RA containing the task identifier (θ_i), the input values (e.g. $(3, 6, 3)$) and the function to be performed (e.g. $\lambda\langle x, y, z\rangle \mapsto \langle x + y + z\rangle$. In practice, this would be implemented by a binary executable that is transfered (along with the data) to the resource on which execution is to take place.

$$AA \rightarrow RA : \textit{task}(\theta_i, \langle 3, 6, 3\rangle, f)$$

Assuming the RA accepts the task it will execute the function and, when it completes, send a message to the AA containing the task identifier and the output data.

$$RA \rightarrow AA : \textit{results}(\theta_i, \langle 12\rangle)$$

An agent executes its task graph by selecting a runnable task and allocating it to a resource (see figure 2). The first clause (π_1) applies when there exists a runnable task. This clause selects a runnable task and allocates it. The second clause (π_2) applies when there are no runnable tasks. This clause waits for a task to become runnable and then continues with execution (which will allocate it).

The agent has plans to allocate a task. Firstly, we check whether the task is similar or identical to others. If it is (π_3,π_4), then we construct a merged task and allocate it (see figure 3). Allocation (π_5) is done by sending a message to the broker. The response by the broker is processed by further plan clauses (figure 4).

Once the AA has received from the broker agent a list of potential resources (Rs) to which the task can be allocated it needs to select a resource. This list will only be sent by the broker if the resources are currently available, and have the capacity to execute the task. It may be possible, for instance, for a resource to be particularly suitable for running tasks of a particular type (or be the only resource with the capability to execute such tasks) but not be available at the time the resource request is made. The broker agent must therefore determine whether to return resources currently available, or the best matching resources. A number of possible strategies may be adopted:

1. The AA selects a resource from the list at random, and sends it a request. The resource is free and accepts the allocation. The task is transfered to the resource, and execution of the task commences.

2. None of the resources on the list are able to accept the allocation at the present time, i.e. all are busy. In this case there are a number of strategies that the AA can pursue:

   - The AA could submit a new request to the broker with weaker requirements (π_7, where we denote a request with weaker requirements by *weaken*(θ_i)). For example, it could indicate that it is willing to accept compute servers with less memory.
   - The AA could ask the broker to send it a list of all resources which match its criteria – even those that are not currently available (this is denoted by plan π_8). Based on this list, the AA determines which (other) AA owns the task that is keeping the resource busy, and requests the AA in question to release the resource. This interaction involves a negotiation between the AA that wants a resource, and one that currently owns the running task on the resource.

This behaviour is described by the plans in figure 4. In the scenario where AA_2 wishes to execute a task θ_i^2, on a given resource RA_j, which is currently running task θ_k^1 (for AA_1), it is possible for AA_2 to:

1. Request RA_j to preempt θ_k^1 in preference for θ_k^2. Based on the belief set for RA_j, and its current plans, either the existing schedule on RA_j is aborted, or the request from AA_2 is ignored.

2. Request AA_1 to preempt its task on RA_j, and reserve this resource for AA_2. If AA_1 agrees, it will make a preemption request, and pass a reservation token to RA_j to enable AA_2 to then schedule its task on RA_j. It is now up to RA_j to accept or deny the reservation request from AA_1

   This assumes that agents are altruistic, or at least cooperative. Note that although conflict (in the allocation of resources) is usually cast as being between two agents, conflict can also arise *within* a single agent if that agent has more than one task graph. In this case we certainly can assume cooperation.

3. Request a higher priority level from a broker agent (B2), and use this as a means to pre-empt a task from another application agent. This strategy would not require a direct interaction between AAs in order to resolve conflicts.

Figure 5 illustrates these interactions – and demonstrates the case where a negotiation between AA_1 and AA_2 takes place to abort task θ_k^1 on RA_1. The release/reserve protocol for resource RA_j is as follows:

1. $AA_2 \rightarrow B$: Request(B, Capability(RA))

2. $B \rightarrow AA_2$: Reply([RA_1])

3. $AA_2 \rightarrow AA_1$: Request(AA_1, Abort(θ_k^1, RA_1)), where Select(AA_2, RA_1) $\wedge \neg Available(RA_1)$

4. $AA_1 \rightarrow RA_1$: Request(RA_1, Reserve(θ_i^2))

5. $AA_1 \rightarrow AA_2$: Reply(AA_2, Abort(θ_k^1, RA_1))

6. $AA_2 \rightarrow RA_1$: Request(RA_1, Schedule(θ_i^2))

3.2 Dealing with conflicts

In the context of such a BDI system, it is possible for agents to have conflicting sub-goals in order to satisfy their overall goal of minimising execution time, or maximising utilisation. Conflicts can arise between plans within an AA or an RA. For instance, an AA may try to group tasks to minimise execution times – however, the new grouping may not be runnable on the available resources, resulting in the agent having to find an alternate grouping, or to wait until the required resources are available. In this case, the grouping of tasks achieves the subgoal of combining tasks with common properties, but it violates the global goal of minimising the execution time.

$\pi_1 =$ exec : task(θ_i, In, Out, f) $\wedge$ runnable(θ_i) $\leftarrow$ allocate(θ_i) ; exec
$\pi_2 =$ exec : **otherwise** $\leftarrow$ **waitfor** task(θ_i, In, Out, f) $\wedge$ runnable(θ_i) ; exec

Figure 2: Exec plans

$\pi_3 =$ allocate(θ_i) : identical(G) $\wedge$ $\theta_i \in$ G $\leftarrow$ mergeIdentical(G,θ_{new}) ; allocate(θ_{new})
$\pi_4 =$ allocate(θ_i) : similar(G) $\wedge$ $\theta_i \in$ G $\leftarrow$ mergeSimilar(G,θ_{new}) ; allocate(θ_{new})
$\pi_5 =$ allocate(θ_i) : **otherwise** $\leftarrow$ send query θ_i to broker agent

Figure 3: Allocate plans

Similar conflicts can arise in RAs trying to optimise their local schedule to improve the overall utilisation of the resource.

When modelling conflicts, it is important to relate the achievement of a sub-goal with the global goal that the agent is aiming to satisfy. The general case is that the agent should not try to attempt a sub-goal which conflicts with its global goal, based on the information that the agent has about the environment at any time. In the context of task and resource allocation, this relates to violating the global goals of minimising execution time (for AA) and maximising resource utilisation (for RA). The point at which a conflict is detected between the sub-goal and the overall goal of the agent determines the "dynamicity" of the agent. For instance, an agent may initiate a plan to achieve a sub-goal, but the environment may change resulting in the beliefs of the agent changing, and the sub-goal conflicting with the global goal. In this case, a dynamic or "cautious" agent would abandon the plan, and search for plans that match its current beliefs. A static or "bold" agent on the other hand would only detect a change in the environment once it has completed executing the current plan, and then determine which other plans become valid. The use of either strategy depends on the rate of change of the underlying environment, and the time overhead of abandoning the current plan and choosing a new one. Kinny and Georgeff [Kinny and Georgeff, 1991] and subsequently Schut and Wooldridge [Schut and Wooldridge, 2000] define this as the "degree of boldness" of an agent – which represents the maximum number of plan steps the agent executes before re-considering its intentions. Their work however has focus on agents which operate in environments which are simpler than the ones we outline here.

In a general case, we can model conflicts between plans that achieve sub-goals, and the global goal as $Con(\alpha_{sub}, \alpha)$, where the number of sub-goals can vary. A dynamic agent checks for this conflict after executing each action within a plan, provided that the rate of change of the environment does not exceed the rate at which the agent can achieve its intentions. Aborting a plan (*Abort()*) implies that the agent abandons the next activity within the current plan, and tries to search for another plan that matches its current beliefs. Hence, executing a plan is equivalent to

$$\forall a \in \pi_i, Exec(\pi) \rightarrow Exec(a, tail(\pi))$$

We must determine ways to (1) detect conflicts, (2) ways to deal with conflicts. We can detect conflicts by analysing changes in the properties associated with an application or resource. There may be a number of possible types of conflicts which may arise within each agent. Conflicts can be between two goals, denoted $Con(\alpha_{sub}, \alpha)$ – in which case the plan that leads to the sub-goal α_{sub} is aborted

$$\pi_i(X, \alpha_{sub}) \wedge Con(\alpha_{sub}, \alpha) \rightarrow Abort(\pi_i)$$

provided α is the goal of higher importance to the agent. A conflict between a goal and a plan is treated in the same way, leading to the plan being aborted. In the context of Computational Grids, this can arise when a RA tries to improve utilisation by running the longest running task, although this task may not utilise the capability available at the resource. In this scenario, other tasks which cannot run elsewhere may need to wait for the resource to be released. We can detect goal conflicts by evaluating the changes that two goals would make to the properties of a resource or application.

If a conflict between two plans arises, then the agent must decide which plan to pursue with the current beliefs. This scenario is particularly important in the context of dynamic (cautious) agents, where the beliefs of an agent may change to make the currently conflicting plan more viable. We denote this as $Con(\pi_i, \pi_j)$, and the corresponding sub-goals as α_i and α_j. We can detect conflicts between plans by evaluating changes that the actions within a plan would make to resource or application properties.

4 An Example

Consider a system consisting of the following named resources:

Zeus (Z): An application server

Apollo (A): Another application server

Vulcan (V): A compute server with multiple CPUs, and with capability to run graphics tasks

Mercury (M): A computer server with a single CPU, and with capability to run database tasks

Hercules (H): A slow compute server

Zeus has a simple task graph consisting of three tasks ϕ_1, ϕ_2, and ϕ_3 where ϕ_3 cannot be scheduled until both ϕ_1 and ϕ_2 have been completed, but ϕ_1 and ϕ_2 can be scheduled in parallel. Apollo has a task graph consisting of four tasks ϕ_4 through to ϕ_7 which can all be scheduled in parallel. The nature of the tasks is such that all tasks can be scheduled on any machine, but ϕ_3 is a database task, and will be particularly slow to complete unless scheduled on Mercury. Similarly, ϕ_4 and ϕ_5 are graphics tasks

```
$\pi_6$ = brokerResponse($\theta_i$,Rs) : R = head(Rs) $\wedge$ Rs $\neq$ [ ] $\wedge$ task($\theta_i, In, Out, f$) $\leftarrow$
    send to R : task($\theta_i$, $\langle V_1, \ldots, V_n \rangle$, $f$) ;
    receive task($\theta_i$, Status) ;
$\langle V_1, \ldots, V_n \rangle$ are the values corresponding to $\langle L_1, \ldots, L_n \rangle (= In)$
    **if** status $\neq$ ok **then** brokerResponse($\theta_i$, R = tail(Rs))
$\pi_7$ = brokerResponse($\theta_i$, Rs) : Rs = [ ] $\leftarrow$ send query *weaken*($\theta_i$) to broker agent
$\pi_8$ = brokerResponse($\theta_i$, Rs) : Rs = [ ] $\leftarrow$ send query $match(R, \theta_i) \wedge \neg Available(R)$
$\pi_9$ = brokerResponse($\theta_i$, Rs) : Rs = [ ] $\leftarrow$ send message("No Resource Available")
```

Figure 4: Plans to allocate a task to a resource using the broker's response

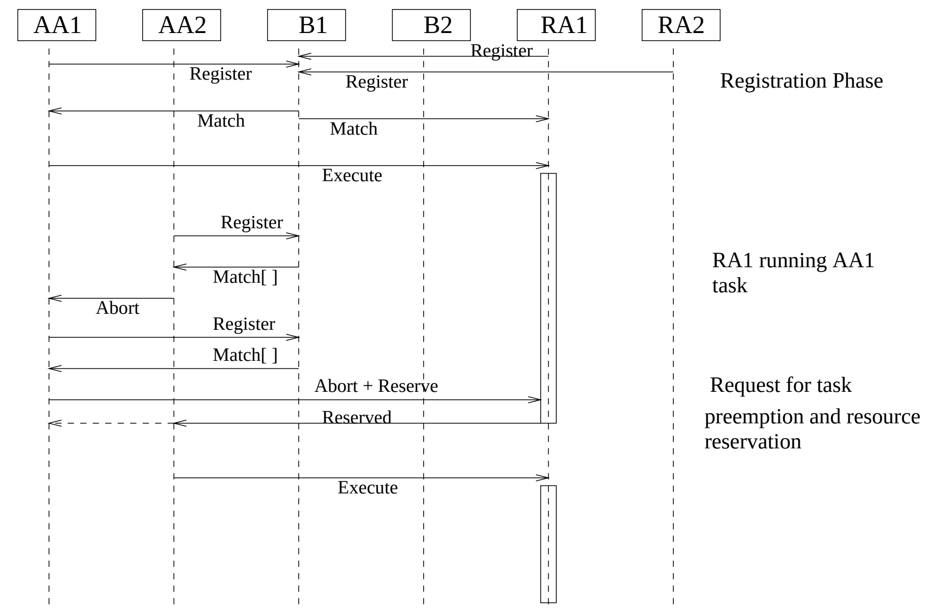

Figure 5: *Interaction between Application (AA) and Resource (RA) agents, mediated by Broker (B) agents*

that would complete quicker on Vulcan. Hence, the global goals include:

$$Goal(A) = min(\sum_{i=1}^{3} exec(\phi_i) + exec(\pi))$$

with a similar formulation for Goal(Z), and,

$$Goal(V) = max(\sum_{(p=CPU,Memory)} C(p) + exec(\pi))$$

which also holds for Goal(M) and Goal(H). Each agent must now monitor its properties to determine which plans apply. Figure 6 provides one execution sequence for tasks $\phi_1 ... \phi_7$. In this example we assume that each resource agent can only execute a single task at a time. However, this constraint does not invalidate the the more general condition where a resource can execute multiple tasks simultaneously. All application and resource agents register with the broker (B) to start with. Application agents send a record for each task that needs to be executed. Resource agents send a record of their capabilities to B. As all resources can run all tasks, B sends a list of all task records to all resource agents, and all resource records to both application agents. Z then requests V to execute ϕ_1. However, as V has better capability to run ϕ_4, ϕ_5, V ignores the request, and instead asks A to submit ϕ_4 or ϕ_5. Z then re-submits its request to M, where it is granted (as there are, currently, no tasks requiring database capability to be executed). Subsequently, A sends ϕ_4 to V, and then ϕ_5 to H. B always maintains beliefs about all tasks that are still waiting to be completed, as indicated in the right hand side of the diagram. After completing a task, each resource agent (V, M or H) re-registers with B to indicate its availability.

4.1 Analysis of system

The system presented here assumes that each agent makes independent decisions about the best way to achieve its goal. There may be conflicts between:

- Goals of various resource agents – as each agent is competing for tasks
- Goals of various application agents – as each agent is competing for resources
- Goals of application and resource agents – as each is aiming to satisfy an objective, that could have conflicting outcomes on the properties of AAs and RAs
- Plans within a resource agent – trying to determine whether to select a task currently available, or wait for one which utilises its capability

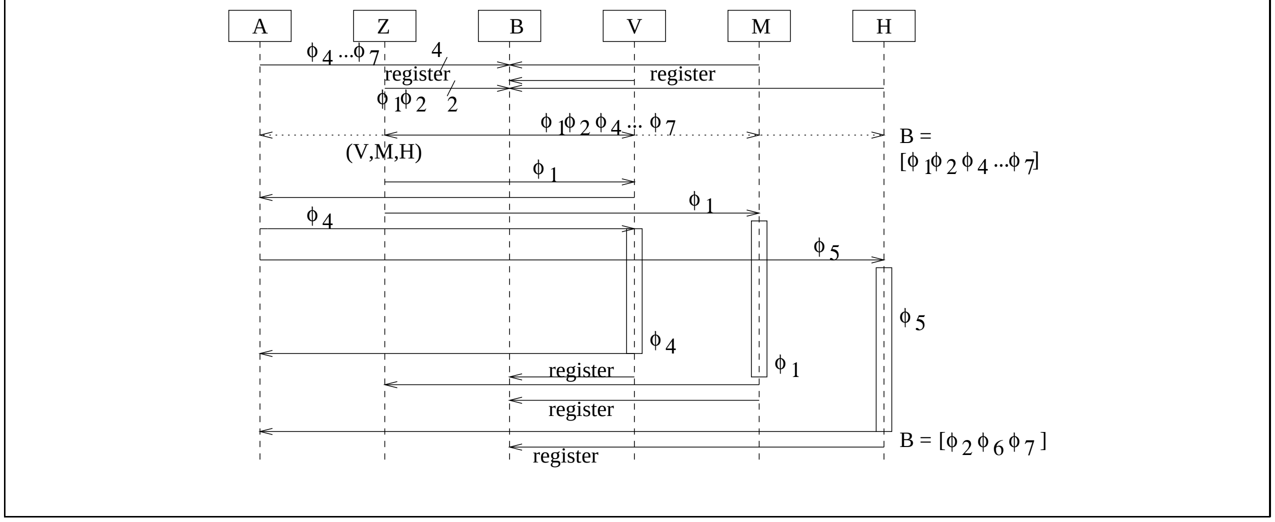

Figure 6: *Interaction between Application (A,Z), Resource (V,M,H), and Broker (B) agents*

- Plans within an application agent – trying to determine whether to run tasks separately, or to group them for a faster CPU resource (for instance)

We outline roles that agents could undertake within such a system. In the examples described here, the broker agent acts as a MatchMaking service, to find similarities between task descriptions provided by AAs and resource capabilities provided by RAs. There is a single criteria over which a match is being achieved. Within the present system we can also include brokers which evaluate multiple criteria to resolve conflicts and choose suitable resources (or tasks):

- Least loaded CPU
- CPU likely to be loaded least after a particular time
- Most available memory (or likely to be available)
- Mean task execution time over entire application
- Most secure (trusted) resource
- Least expensive resource
- Quickest time within deadline threshold

some of these criteria are dependent on the broker executing a prediction algorithm. Our system can easily accommodate these additional criteria, as this would translate to altering the Goal of each RA and AA, based on input it receives from the broker. We could also have multiple broker agents, where each broker either specialises in a given criteria, or uses a different weighting function to rank each of the above criteria.

Figure 7 illustrates a scenario involving task preemption, and is similar to that in figure 6 except that A and Z negotiate to preempt a task on a given resource. In this scenario, agent A negotiates with Z to abort its task ϕ_1 to enable it to execute its task ϕ_6, which it considers to have a tighter completion deadline. Based on its beliefs, agent Z must decide to honour the request, and to initiate the removal of its task from resource M. Agent Z makes a request to abort its task from resource M and make a reservation for A. It is now the decision of resource agent M to honour the request from agent Z. As both Z and M are autonomous, with their own beliefs and plan library, they must make decisions locally. In this case, M agrees to abort ϕ_1, and sends an acknowledgement to A and Z – and does not register again with B. However, M informs B that ϕ_1 has been aborted, and must be re-executed (the beliefs of B now contain ϕ_1). A now sends its task ϕ_6 to M, where execution can start. In this scenario the ability to abort a task resides with the task owner, and the resource on which the task is executing. When agent A requests agent Z to abort its task, agent Z evaluates the priority of the request and makes a subsequent decision. We may also consider such decisions to be supported via another broker agent (B2), which enables application agents to request a priority level for their tasks. B2 may approve priority requests, or may decide on a level different from the request, leaving it up to the application agent to accept this.

4.2 Related work

Support for handling resource capabilities already exist is some metacomputing systems, such as Globus [Globus, 2001] and Legion. The Globus system provides a Resource Specification Language (RSL) to define resource properties and the location of software executables. Two new protocols – the Grid Resource Information Protocol (GRIP) and Grid Resource Registration Protocol (GRRP) are aimed at providing support for discovering new information services, and registering new services with the Globus directory service (the MDS) [Foster, 2001]. Subsequently, a Globus Resource Allocation Manager (GRAM) manages access to a set of resources with the same site-specific allocation policy, where a resource can range from a tightly coupled parallel computer, a cluster of workstations, a data storage system or a scientific instrument. Furthermore, resource ensembles can be managed by a third party system, such as Codine/GRIDWare or LSF [Baker, Fox and Yau, 1996]. In Globus, a Resource Broker is responsible for resource discovery within each administrative domain, which works with an Information Service, and a Co-allocator for monitoring the current state of resources, and managing an ensemble of resources respectively.

The Legion [Natrajan, et al., 2001] system describes resources and tasks as a collection of interacting objects, where compute resources are abstracted as ‘Host’ objects, and data resources as ‘Vault’ objects. The Legion system also provides a set of core objects, that enable arbitrary naming of resources based on Legion Object Identifiers (LOIDs) and Legion object addresses (LOA). Specialised services, such as Binding agents and Context objects are provided to translate between an arbitrary resource name and its physical location – enabling the resource discovery to be abstracted as a translation mechanism between LOIDs and physical resource locations. The Legion system provides a notation for defining resources, based on an object-oriented type system, supporting inheritance and

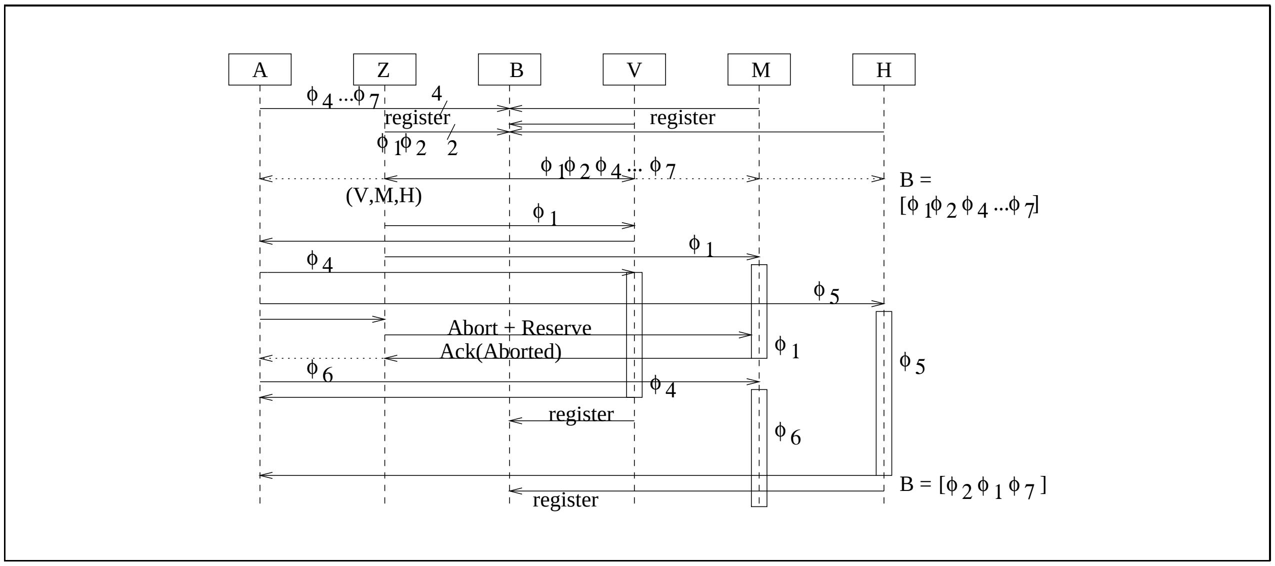

Figure 7: *Interaction between Application (A,Z), Resource (V,M,H), and Broker (B) agents – scenario II involving task preemption*

encapsulation. Legion supports site autonomy with a Jurisdiction Magistrate (JM), which can reject requests that conflict with the policy of a managed site.

Other systems such as Jini [Edwards, 1999] from Sun Microsystems makes use of a Discovery, Join and Lookup service, to enable devices to dynamically enter and leave a cluster. Although in many ways similar to our approach, the discovery protocol supported involves lookup on data types and, in some instances, the class hierarchy (derived type). Any additional matching which exploits associations between device types is not supported in Jini. The TSpaces project from IBM [TSpaces, 2001] also uses data types to seek a match between a service description and a service request. However, TSpaces does claim to provide advanced tuple matching capabilities, which involves conjunction/disjunction of capabilities expressed as data types. Serafini et al. [Serafini et al., 2001] present an approach to optimising queries for data Grids, involving four kinds of agents: user agents, index agents, mass storage agents and “internal” agents. Whereas the first three of these provide wrappers to existing systems, “internal” agents act as query optimisers at different levels of specialisation and criteria. They also suggest an implementing based on BDI agents using JACK [JACK, 2001]. Their work can be easily integrated with our work on Computational Grids, by utilising brokers which hand-over control to their “internal” agents for managing data resources.

Our approach uses the object-oriented description mechanism in Legion, but is most closely related to the approach taken in the ‘class advertisement’ mechanism in Condor [Frey et al., 2001]. In this system, resources describe their capabilities as an advertisement, which is subsequently matched with an advertisement describing the needs of an application. Each advertisement carries a ‘Constraints’ and ‘Rank’ keyword, which must evaluate to True, for a match between a resource and task advertisement to be successful. Matching between resource capabilities and task requirements are based on a classification scheme, which divides resources into one of four categories, (1) a Storage resource, (2) a Computational resource, (3) a Visualisation resource, (4) a scientific Instrument. The proposed approach can utilise Jini based services such as transaction support, leasing etc, and at a minimum support the type matching mechanism supported in Jini. We feel the proposed approach therefore compliments and extends vendor based approaches such as Jini and TSpaces.

5 Conclusion

A BDI approach to modelling behaviours of resource and application agents is presented – and plans for participating agents are described. The use of BDI behaviours enables new application or resource agents to be added, leading to existing agents adapting their behaviours. We believe that the BDI model is most appropriate because it enables each agent to model a site specific administrative policy (for both RA and AA agents), and Broker agents may undertake different plans based on their priorities. The BDI model is also useful in that it can allow agents to enter/leave the environment dynamically, and for all other participants to adjust their plan libraries accordingly.

Various roles that intermediate broker agents could undertake within such a system are outlined, and we also suggest how role specialisation can be used to overcome conflicts. A prototype of this system is currently being developed in AgentTalk (an implementation of AgentSpeak [Rao, 1996]). In subsequent work, we also aim to explore the relationship between $t_{deliberate}$ and $t_{execute}$ as described in section 3.1, and as investigated under the general term of ‘bounded optimality’ in agent systems. The effective control of time to reason is important in this context, as a BDI agent must deliberate only for as long as is necessary – and dependent on environment complexity and agent knowledge/predictability about its environment.

References

[Jennings and Wooldridge, 1998a] N. Jennings and M. Wooldridge. Applications of intelligent agents. In Jennings and Wooldridge [Jennings and Wooldridge, 1998b], chapter 1, pages 3–28.

[Jennings, 2001] N. R. Jennings. An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4):35–41, 2001.

[Georgeff and Rao, 1998] M. Georgeff and A. Rao. Rational software agents: From theory to practice. In Jennings and Wooldridge [Jennings and Wooldridge, 1998b], chapter 8, pages 139–160.

[Kinny and Georgeff, 1991] D. Kinny and M. P. Georgeff, ‘Commitment and effectiveness of situated agents’, in Proceedings of the International

Joint Conference on Artificial Intelligence, pp. 82–88, Sydney, Australia, (1991)

[Frey et al., 2001] J. Frey, T. Tannenbaum, M. Livny, I. Foster, S. Tuecke, “Condor-G: A Computation Management Agent for Multi-Institutional Grids”, Proceedings of the Tenth International Symposium on High Performance Distributed Computing (HPDC-10), IEEE Press, August 2001

[Foster, 2001] I. Foster, “MDS2”, paper submitted to the Grid Information Services Working Group, as part of the Global Grid Forum, March 2001

[Natrajan, et al., 2001] A. Natrajan, M. Humphrey, A. S. Grimshaw, “Capacity and Capability computing in Legion”, Proceedings of International Conference on Computational Science, May 2001

[Jennings and Wooldridge, 1998b] N. R. Jennings and M. J. Wooldridge, editors. *Agent Technology: Foundations, Applications, and Markets*. Springer, 1998.

[Tidhar et al., 1998] G. Tidhar, C. Heinze, and M. Selvestrel. Flying together: Modelling air mission teams. *Applied Intelligence*, 8(3):195–218, May 1998.

[Rao and Georgeff, 1992] A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In C. Rich, W. Swartout, and B. Nebel, editors, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 439–449, San Mateo, CA, 1992. Morgan Kaufmann Publishers.

[Bratman, 1987] M. E. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.

[Baker, Fox and Yau, 1996] Baker, M. A., Fox, G. C. and Yau, H. W. (1996). Review of Cluster Management Software. *NHSE Review*, 1(1), May 1996.

[Edwards, 1999] W. Keith Edwards, “Core Jini”, Addison Wesley, 1999.

[Foster and Kesselman, 1999] I. Foster, and C. Kesselman, (1999). *The Grid : Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers

[IRS, 2001] Intelligent Reasoning Systems. JAM Agent. See web site at: `http://members-http-3.rwc1.sfba.home.net/marcush/IRS/`. Last visited: July 2001.

[JKQML, 1999] IBM Research. A KQML implementation in Java, 1999. See web site at: `http://www.alphaworks.ibm.com/tech/jkqml/`. Last visited: July 2001.

[JACK, 2001] Agent Software Limited. JACK, 2001. See web site at: `http://www.agent-software.com`. Last visited: July 2001.

[Globus, 2001] Argonne National Laboratory. The Globus Systems. See web site at: `http://www.globus.org/`. Last visited: July 2001.

[PRS, 2001] SRI. PRS-CL: A Procedural Reasoning System. See web site at: `http://www.ai.sri.com/~prs/`. Last visited: July 2001.

[Rana et al., 2001] Rana, O.F., Bunford-Jones, D., Walker, D.W., Addis, M., Surridge, M., and Hawick, K. (2001) Resource Discovery for Dynamic Clusters in Computational Grids. In *Procedings of Heterogeneous Computing Workshop*, at IPPS/SPDP, San Francisco, California, April 2001, IEEE Computer Society Press.

[Serafini et al., 2001] L. Serafini, H. Stockinger, K. Stockinger, and F. Zini. Agent-Based Query Optimisation in Grid Environment. In Proceedings of the IASTED International Conference on Applied Informatics (AI 2001), Innsbruck, Austria, February 2001.

[Thanagarajah, 2000] J. Thangarajah. Representation of goals in the belief-desire-intention model, 2000. Honours thesis, RMIT University, Melbourne, Australia

[TSpaces, 2001] IBM Research, “TSpaces: Intelligent Connectionware”, see Web site at: `http://www.almaden.ibm.com/cs/TSpaces/`. Last visited: July 2001.

[Rao, 1996] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. Technical Note 64, Australian Artificial Intelligence Institute, Feb. 1996. Also appeared in Agents Breaking Away (LNAI 1038, p42-55).

[Winikoff, Padgham and Harland, 2000] M. Winikoff, L. Padgham, and J. Harland, (2000). *Conflict in BDI Agent Systems: Taxonomy and Language Constructs*. RMIT University, Melbourne, Australia

[Schut and Wooldridge, 2000] M. Schut and M. Wooldridge, Intention reconsideration in complex environments, Proceedings of International Conference on Autonomous Agents, Barcelona, Spain 2000

[Wooldridge, 2000] N. Wooldridge, (2000). Chapter 2 in *Reasoning about Rational Agents*. MIT Press. ISBN: 0-262-23213-8.