Specification of Coordinated Agent Behavior (The SimpleTeam Approach)[1]

October 1999

Technical Report 99-05

By:

Andrew Hodgson, Ralph Rönnquist & Paolo Busetta
Agent Oriented Software Pty. Ltd., Melbourne, Australia
{ash,ralph,paolo}@agent-software.com
http://www.agent-software.com

[1] This article first appeared in the proceedings of the IJCAI '99 Workshop on Team Behavior & Plan Recognition pp 75-81. It has been edited for clarity and updated to reflect the current JACK Intelligent Agents distribution

Abstract

‘Team oriented programming’ indicates a number of different approaches to the formation of teams of agents and their coordination in order to achieve common goals. A common characteristic of these approaches is that the activity involved is seen from the abstract point of view of the team as a whole. This paper presents a framework, called SimpleTeam, aimed at team oriented programming. SimpleTeam is an extension to an existing Java-based multi-agent framework, JACK Intelligent Agents™, which supports the Belief-Desire-Intention (BDI) agent architecture.

SimpleTeam supports the writing of team plans, which represent the activity of a group of agents or sub-teams in order to achieve a team goal, and provides a set of primitives to control concurrency, exception handling and so on. Team members are referred within a plan by their role, which is associated to a specific agent during the formation of the team. Team formation and team plan execution is controlled at run-time by two special types of agents called team manager and team instance. SimpleTeam does not commit to a specific approach to the formation of teams and distribution of beliefs; it can be developed by the programmer to follow the paradigm most appropriate to the task at hand. In the specific domain of agent-based simulations, SimpleTeam enables the programmer to choose the fidelity level of a simulation of a group by allowing the ’collapsing’ of single entities within the team instance.

This paper introduces the software architecture of SimpleTeam, discusses its most important concepts and shows some of its primitives. An example application is presented.

1 Introduction

‘Team Oriented programming is a nuance of Agent Oriented programming wherein agent collaboration is specified from the abstract view-point of the group as a whole. The concept behind this approach is that coordinated behaviour is specified, i.e., programmed, from a high-level (’bird's-eye’) perspective and that the underlying machinery maps such specifications into the individual activities of the agents concerned.

Within Artificial Intelligence research, team work as an agent programming activity has been studied since the early 90s (Cohen 1991), and is a rapidly developing field. The objective is to find a general-purpose model of team work which simplifies the implementation of any particular mode of coordination.

JACK Intelligent Agents™ is a Java-based multi-agent framework developed by Agent Oriented Software Pty. Ltd. JACK enables the development of complex agents in Java and supports the Belief-Desire-Intention (BDI) architecture (Rao & Georgeff 1992). It is further designed to allow extensions that implement new or extended agent models through a plug-in paradigm with the core JACK framework.

JACK allows for the development of robust intelligent agent systems following the BDI agent model. The BDI model is applicable to a wide range of applications and can be extended to introduce team work concepts. In particular, modeling principles regarding specifications of teams (Cavedon et al,1997, Tambe,1997) and schemes for centrally specifying multi agent dialogs/protocols (Haddadi1996) can be introduced. We believe that these abstractions are valuable for developing complex systems involving many collaborating agents.

In this paper, we suggest a plug-in extension to JACK that allows for the specification of simple teams and the coordination of joint activities among the team members. We then focus on providing the most valuable and practical aspect regarding teams; the centralized specification of coordinated behavior, and its realization through actual coordinated activity. The suggested extension is implemented as a JACK plug-in without modifying the core. It results in an addition of new concepts, and as an extension of the JACK capability concept to abstract the definition of behavior (role) from the implementation (capability).

Following the JACK paradigm, team concepts are brought in as a strictly typed language. This is an accepted practice within mainstream Software Engineering, as it allows early detection of inconsistencies during the compilation and initialization process. In order to specify coordinated behavior in a type-safe way, we have introduced a number of new entities (concepts) to JACK. Their role is to provide a consistent scheme for specifying the abstract behavior of teams, specifying the coordination of activities between the various components of teams, and providing an infrastructure for instantiating particular instances of teams.

Many different theories and types of teams, ranging from strictly hierarchically structured teams to collaborating teams without formal structure, have been proposed in the literature. Also, theories have been proposed regarding mutual beliefs and goals, where individual members of a team attempt to achieve what they believe the team as a whole is attempting to achieve.

The work presented here is neutral to the nature of the structure of a team (i.e., hierarchical and imposed ’from the top’, or resulting from spontaneous collaboration ’from the bottom’), and to how team formation is achieved. Our only assumption is that, after formation, it is possible to classify the members of a team in terms of abstract roles. Our goal is to provide a software infrastructure for the specification of coordinated behaviour which can then be used for pursuing applied studies on social organization.

2 Team Based Architecture

Among the goals we set out to achieve, two are worth mentioning before introducing the proposed software architecture.

The first is to reduce the inherent complexity in design, computation and communications that arises from attempting to implement fully distributed control of coordinated behaviour. A simple example is addressing when sending a message to a team as a whole instead of a specific agent; for instance, in a military situation, a commander may wish a command to include the entire platoon, not just one soldier.

The second goal is strictly related to a specific domain of application; that is, simulation of social entities (for example, military organizations such as platoon, brigades, and armies). Very often, the level of fidelity required may make the simulation of the single entities composing a team (soldiers and commanders in the example above) is excessive, while a coarse-grained view is sufficient. For instance, it may be sufficient to know that a platoon moved from one location to another and that it suffered a certain percentage of losses during the maneuver. At the same time, we want to be able to use only one language for specifying the coordinated behaviour of a team, no matter how this will be simulated: by any number of agents; by one per member of the team; or by a single entity in the future.

The easiest, and most obvious, way to achieve these goals is the introduction of a new software entity within a team-based multi-agent system. We call this new entity 'team instance'. A team instance is the software representative for a team, and its goal is to coordinate the behaviour and facilitate the communications of zero, one or more agents; or, recursively, other team instances, each representing single elements or further aggregations within the team.

Thus, in pure software terms, a team has a centralized ('star') organisation, independent of the structuring and internal dynamics of the social organisation being represented. All communication and decision-making related to coordinated behaviour happens by means of the team instance. Teams can be created dynamically during execution; moreover, agents and teams can be part of multiple teams at the same time.

Team instances can be compared with the so-called facilitators which are commonly used in multi-agent systems (see, for instance, Open Agent Architecture [Cohen 1994). However, while facilitators normally provide only a sophisticated communications and brokering facility, team instances actually perform all reasoning related to coordination and distribution of activity among members. As we discuss in the next paragraphs, team instances can even simulate the behaviour of team members that have not been instantiated.

3 Team Manager

One of the more complex issues concerning team modelling is what may be called the 'Team Formation' issue. That is, how to specify the way in which particular teams come about, or how particular sub-teams and agents become members of particular teams.

For the discussion in this paper we have chosen largely to side-step the team formation aspect and leave it to the programmer. As an aid, there is a Team Manager, which is an entity (a JACK agent) that keeps track of all team instances and the roles they can adopt. On construction, team and agent instances can register with the Team Manager, explaining their own capacities. Then team instances can request these entities to fill required roles. A base Team Manager behavior is provided through a standard Team Manager capability, and this can be tailored by means of defining new plans.

The team formation process also involves acceptances from presumptive sub-teams to fill particular role entries, and the reasoning underlying these decisions are programmed as sub team plans.

4 The Team Construct

The team construct is the first and, in a way, the most important new concept. It is designed to:

- specify what a team is capable of doing (i.e., what roles it can fulfill);
- specify which components are needed to form a particular type of team;
- specify when/whether the team is willing to take on a particular role within another team;
- specify the coordinated behaviour among the team members; and
- specify some team knowledge.

The team specification determines what each team member does, and it also handles failure of members to achieve their goals. Team members act in coordination by being given goals according to the specification, and they are themselves responsible for determining how to satisfy those goals.

Team knowledge is not an implementation of mutual beliefs (Tidhar et al.1998), but is rather a practical way to keep state information about the team and its activity. In other words, the team members do not have mutual beliefs, but the framework includes a place for shared knowledge in connection with the team specification. Team members can then access this information through standard messaging.

Teams are defined in a way similar to agents in JACK; that is, a team is a class level construct which brings together the various components that form that type of team. It appears as follows:

```
team platoon extends SimpleTeam {
     // Team declarations
}
```

In essence, a team class is an extension to agent taking responsibility for managing the coordination of team member activity and holding the team's knowledge base.

A key element of a team definition is the declaration of which roles this team can play as member of other teams. The JACK statement `#performs` role is used for this, as illustrated by the following example.

```
#performs role GroundSurveillance;
#performs role GuardDuty;
```

Any one team may be declared as willing to take on several roles, and in some situations, multiple roles at the same time.

Each `#performs` role statement is a declaration that the type of team can perform the role. The statement also ensures that instances of the team class register with the team manager as being able to take on the roles mentioned.

The members of a team are declared through role requirements rather than explicit sub-team type requirements. This can be compared with the use of interfaces in Java; it expresses the functional requirements for the members rather than restricting to particular team types. Team member declarations are as follows:

```
#requires role RadioOperator  radio;
#distinct role Soldier[6]     soldiers;
#requires role Medic          medic;
```

Statements like these describe the team in terms of the roles of the sub-teams. The keyword `distinct` is used for roles that are to be distinct; that is, to prohibit the same team instance taking on more than one of the distinct role entries. For instance, the team of the example above requires a `RadioOperator, 6 Soldiers` and a `Medic`. However, only the `Soldiers` need to be distinct. Therefore, an instance that performs all the roles of `RadioOperator, Medic` and `Soldier` could take on all three role entries simultaneously.

Role entries are filled during team formation. The team is then allowed to choose from the set of available team instances that are 'willing' to take on the roles: the actual team formation is done by handling a `TeamFormationEvent` in the team. There is a range of control primitives to use in the `TeamFormationEvent` handling, but these will not be discussed further here.

5 Team Plans

The *team_plan* concept is a variation to the JACK plan used to specify coordinated activity. A team_plan includes the normal plan statements, and has an additional statement form for specifying parallel execution of goals by sub-teams. Further, it admits an extra construct by which to extend the BDI context to include knowledge about the role that the team is undertaking.

Comparing team_plan and JACK plan, there are three distinct differences.

First, the applicability constraints are extended to allow limiting a team_plan to only be applicable for performing a task for some particular role. This can be important, for instance, if you have a team that can take on multiple roles and it is to act differently dependening on which of the roles is 'active'. For example, consider a team that is able to perform two roles named `AerialSurveillance` and `GroundSurveillance`. Both these roles can handle the same type of observation event, but with different plans, and only the plans of the role actually taken on should be applicable.

Second, the body of a team_plan allows for action designators that in effect issues goals for selected team members to achieve. These are introduced by means of a goal statement of the form `@team_achieve( goal )`.

The `@team_achieve` statement is similar to a synchronous sub-task which succeeds or fails with the processing of the goal. That processing, however, is carried out by the designated team member rather than by the team entity itself.

The third, but perhaps most important, aspect of team plans is the ability to specify parallel actions. Consider, for instance, the following action schema:

```
platoon A moves to left flank
and in parallel
platoon B moves to right flank,
then both attack the center.
```

This implies coordination between the actions such that the first two occur in parallel, and the next step is not begun until both (all) of the parallel actions have completed successfully.

We handle parallel actions by introducing a block level construct that processes its sub-statements in parallel. The action schema above could then look as follows in a team_plan:

```
@parallel (...) {
   @team_achieve(A.move("left"));
   @team_achieve(B.move("right"));
}
   @parallel (...) {
   @team_achieve(A.attack("center"));
   @team_achieve(B.attack("center"));
}
```

The sub-statements of the `@parallel` block are performed conceptually in parallel through an interleaving execution which adheres to the statement atomicity principle of JACK. That is, although the computations are executed in parallel, it is a controlled execution which ensures computations to never be interrupted in the midst of a volatile region.

The `@parallel` statement is a proper language statement and can occur as such anywhere within the reasoning methods of team plans. In particular, `@parallel` statements can be nested, as outlined in the following example:

```
@parallel (...) {
   for (i = 0; i < 100; i++)
   @achieve(...)
   @team_achieve(...)
   @parallel (...) {
        ...
   }
}
```

The example results in three parallel computations - the for-loop, the `@team_achieve`, and the inner `@parallel` statement - are all in parallel. In effect of course, the sub-statements of the inner `@parallel` statement end up being performed in parallel with the `@team_achieve` and the for-loop. However, a `@parallel` statement also takes the control arguments relating sub-statement success and failure to overall success and failure. As a result, the `@parallel` statement is thus a statement with a number of variants, such as whether the statement requires all or only one sub-statement to succeed, or whether it succeeds immediately with the first success, etc.

5.1 Variants of Parallelism

There are many useful variants of parallelism. Which is the appropriate one depends on the application. When designing the various types of parallelism we see three attributes that differentiate these:

- Success condition. This attribute specifies when the `@parallel` statement as a whole succeeds. The following variants have been designed:
- All: The `@parallel` statement succeeds when all the parallel sub-statements have terminated successfully. This can be viewed as a parallel AND statement, because all sub-statements must succeed. Further, the `@parallel` statement fails immediately with the first sub-statement failure, and all on-going sub-statements are then notified accordingly (see below).
- Last: The `@parallel` statement succeeds when all the parallel sub-statements have terminated successfully. However, failure is postponed until all sub-statements have terminated.
- First: The `@parallel` statement succeeds as soon as any one of the parallel sub-statements succeeds, and all on-going sub-statements are then notified accordingly (see below). This can be viewed as a parallel OR statement with short circuiting.
- Any: The `@parallel` statement succeeds if one of the sub-statements succeed, but does not terminate until all sub-statements have terminated.
- Termination Condition. This attribute is a triggered condition which will terminate the `@parallel` statement if it becomes true. Ongoing parallel sub-statements are notified but treated as failed, and the `@parallel` statement will succeed or fail according to the success condition. In particular, if the success condition is 'any', and some sub-statement has already succeeded, then the `@parallel` statement as a whole will succeed even if the ongoing sub-statements are terminated due to the termination condition.

- The termination condition can be any triggered expression in JACK and may in particular be affected by some of the sub-statements. For instance, a sub-statement may result in that some relation in the team changes and this could be the condition that terminates the `@parallel` statement.
- Notification Exception. This attribute provides programming control of how sub-statement are notified about termination. The termination takes effect immediately for the `@parallel` statement by means of detaching it from all on-going sub-statements. The sub-statements are thereafter notified by using the given exception. If the value is null the sub-statements are detached and executed to completion without any notification.

It is worth noting that the `@team_achieve` statement propagates the exception to the team member concerned before throwing the exception to its invocation.

5.2 Unstructured Parallelism

In addition to the structured parallelism discussed above, we introduce a means of more detail control of the parallel execution. The idea is simply to treat `@parallel` as an expression that returns a handle on the parallel executions, which then becomes an object that can be probed and manipulated. In code, this can be written as follows:

```
Parallel p = @parallel(...) { ... }
p.execute();

@wait_for(p.substatement(3).finished());

@wait_for(p.finished());
if (p.passed()) ...
```

The handle for the parallel statement can thus be probed regarding the termination and/or success of the individual sub-statements, and the computation can be synchronized by means of `@wait_for` statements. In this way the `@parallel` statement as well as its individual sub-statements can be waited on, terminated, probed etc. The details of this are beyond the scope of this paper.

5.3 Parallelism and Exception Handling

Exception handling within the parallel execution model is designed to strictly follow the Java model for exceptions. That is, a sub-statement may throw an exception and, if not caught, the exception is propagated upwards in the calling stack. When the exception reaches the `@parallel` statement, this causes a notification to any on-going sub-statementbefore the exception propagates out of the `@parallel` statement.

A sub-statement may catch exceptions, as shown in the following example:

```
@parallel(...) {
   try {
     @achieve(...)
     } catch (...) {...}
     finally {...}

   try {
     @test(...)
     } catch (...) {...}
     finally {...}
}
```

In this example there are two parallel sub-statements that each contains a try-catch-finally block. If, for instance, an exception is thrown within the `@achieve` statement, the parallel sub-statement may catch that exception and succeed anyhow.

Exceptions thrown in the handling of `@team_achieve` are propagated in a slightly different way (through message exchange) in particular, since the team member performing the subtask (and throwing the exception) may reside in a process other than the team entity process.

5.4 Result Unification

One of the more difficult issues in dealing with parallelism is how to deal with the possibility of contradictory results of the various sub-statements. This is a common problem that always arises when parallel computations share the data space. There are many ways of approaching this problem with different sorts of compromises being made.

The particular approach we provide concerns the use of logical variables and, in particular, logical variables that are unbound at entry to the `@parallel` statement. In this case, each sub-statement will effectively receive its own copy of the logical variable to use and perhaps bind to some value. As the sub-statements succeed the 'copies' of the logical variables are unified. If any such unification fails, which means that the sub-statement has terminated successfully but bound some logical variable differently than some previously successful sub-statement, then the new sub-statement is treated as failed. Later, when the `@parallel` statement succeeds (according to its success condition), the bindings that are then available remain as result bindings from the `@parallel` statement.

The handling of logical variables is thus well defined, and the programmer is free to express explicit combinations of logical variables.

6 The Role Construct

The role construct is used to specify abstract capability; that is, it is an interface definition, which declares what an entity that claims to implement a role must be capable of doing. This is specified in terms of events handled and posted by the entity that fills such a role entry within a team.

As stated before, our role concept can be compared to the use of interfaces in Java; its purpose is to declare an interface which an entity must implement to fill such a role. In that way, the role construct provides a separation between what a team member needs to be able to do and how it does it.

For example, a role could be implemented by a complex team providing a high level of fidelity or by a simple agent that produces a similar result but without further sub-dividing the task. In such a case, the choice of which entity to use would be dependent on the particular activity. If, for example, there is an interest in how individuals of a platoon perform a task, then a platoon team would be used, but if the interest is merely what happens to the platoon (at a macroscopic level) then programming this as a single agent could be sufficient and more efficient.

A role definition has two parts. Firstly, a 'downwards' interface that declares the events an entity must handle to take on the role, and secondly, an 'upwards' interface that declares the events the team entity needs to handle when having a team member of the role. The following is an example of a role definition:

```
role GroundSurveillance extends Role {
     // the downwards interface
     #handles event Observe obs;
     // the upwards interface
     #posts event Evacuate;
}
```

Note that the 'downwards' events are sub-tasked by the team entity by means of `@team_achieve` statements. The 'upwards' events are instead posted by the role filling entity and handled by the team entity.

7 Team Capability

As with JACK agents, teams in JACK are defined as having named capabilities. The team capability concept is an extension to the agent capability concept. The team capability uses team plans and includes a statement form to declare that the capability implements a role. This is a simple addition to the capability concept, which is written as follows:

```
capability UAVRadar extends Capability {
     #implements role AerialSurveillance ;

}
```

In the example, an agent (or team) that has the `UAVRadar` capability is able to take on an `AerialSurveillance` role in a team. However, the agent (or team) definition must also include the statement

```
#performs role AerialSurveillance;
```

to declare that it is generally willing to fill the role.

8 Example

This section illustrates the use of the team modelling concepts by means of an example. We have chosen to model a software development team, since this is a relatively well understood domain, with several roles, and involving a reasonable amount of parallelism.

A software development team is setup to have the following roles:

- project management (ProjMan)
- quality assurance (QualAssur)
- requirements (Requirements)
- research (Research)
- design (Design)
- coding (Coding)
- testing (Test)
- integration (Integration)

Only some of these roles will be expanded in this example. The overall team is defined as follows:

```
team SoftwareDevelopment extends SimpleTeam {
   #requires role ProjMan      projman;
   #requires role QualAssur    qual;
   #requires role Requirements req;
   #requires role Research     res;
   #requires role Design       des;
   #distinct role Coding[3]    coders;
   #requires role Test         tester;
   #requires role Integration  integrators;
```

```
   #uses plan DevelopSoftware;
}
```

We note that the team has three distinct coders, otherwise the roles are singular. Since only the coders are distinct, other roles may be filled by the same team member or by distinct team members according to the formation plans.

For the Requirements role, we model two alternatives. If the actual project software undertaken is less complex, the requirements can be handled by a simpler team which directly performs the task. For complex projects, a larger requirements team may be needed as modelled by `ComplexRequirements` below.

```
team SimpleRequirements extends SimpleTeam {
   #performs role Requirements req;

   #uses plan getRequirements;
}
team ComplexRequirements extends SimpleTeam {
   #performs role Requirements req;

   #requires role ClientLiason cl;
   #requires role TechLead     lead;
   #requires role Documentor   doc;
   #requires role Research     researcher;

   #uses plan AcquireRequirements;
}
```

The larger requirements team thus includes an explicit role separation for client liaison, technical leadership, documentation and research.

We continue the illustration by suggesting a team plan for the SoftwareDevelopment team according to the following:

```
team_plan DevelopSoftware extends TeamPlan {

   #uses team SoftwareDevelopment team;

   body()
   {
      @team_achieve(team.qual.EstablishProcedures());
      @parallel()
      {
         @team_achieve(team.projman.Manage());
         @team_achieve(team.qual.EnsureQuality());
      }
      @parallel()
      {
         @team_achieve(team.req.getRequirements());
         @team_achieve(team.res.findBestTech());
      }
      @team_achieve(team.qual.checkReqDoc());
      @parallel()
      {
         @team_achieve(team.des.developArch());
         @team_achieve(team.coders[0].prototypeGui());
         @team_achieve(team.coders[1].prototypeDb());
         @team_achieve(team.coders[2].prototypeComms());
      }
      @parallel()
      {
         @team_achieve(team.qual.checkArch());
         @team_achieve(team.req.reviewProtoTypes());
```

```
        }
        @team_achieve(team.des.developDesign());
    }
}
```

The reader may verify that the team plan represents a waterfall development model, with some stages broken down into individual tasks. For instance, the architectural stage involves prototype building for the user interface (`prototypeGui`), the back-end data repository (`prototypeDb`), and the communications infrastructure (`prototypeComms`). The prototype building occurs in parallel with the architectural design (`developArch`).

We note also that the plan includes a declaration which provides access to the team structure. The team plan is a plan for the team entity with the purpose of coordinating the team member activities.

From this example (although far from complete) we can highlight some features of the modelling approach and also point out some short-comings:

- It allows for the description of coordinated activities in a clear and centralised way.
- It allows the abstraction of what needs to be done from how it is done, and allows for the team plan to be written without consideration of how the roles are fulfilled. This is highlighted by having two very different teams that can take the role Requirements.
- It shows how rapidly even relatively simple team programs become complex. Building a robust team application requires good software engineering practices and tools, including team modelling.

We note that developing the same application without team modelling concepts: - developing the plans for individual agents and messages - would easily result in an unmaintainable system. A change to the team behaviour would then impact many agents, and the centralised specification would be lacking.

At the same time, although the example presents a realistic team structure and flow of activity, it implements an idealised view that may not always be valid. For instance, as project management may run in parallel with other activities, there may also be intricate control structures across the various parallel activities (e.g., weekly reporting, etc.). It is not immediately clear whether the modelling approach allows such control to be captured in an easy and natural manner.

9 Conclusion

We have presented an introduction to team modelling concepts as an extension to the JACK Intelligent Agents framework. It is a powerful scheme specifying and implementing coordinated distributed behaviour in a manner which maintains type-safeness and generally follows the JACK and Java paradigms.

The team modelling concepts have been defined as a plug-in extension to JACK, and as such, it illustrates how the built-in adaptability of JACK can be utilised to cater for extensions and variations to the modelling infrastructure.

By focusing on coordinated activity as the main aspect of team modelling, we have obtained a limited but well-defined modelling capability that can be implemented efficiently and relatively easily. At the same time, it is a limited model that is applicable only to certain types of collaboration, wherein particular team structures are rigid and pre defined. Whilst being useful for a range of applications, there is also room for further work in several areas, such as:

- allowing for additional types of parallelism
- dealing with dynamic team formation and re-organization
- allowing different types of teams

- including mutual beliefs and goals

We expect some of these aspects to fit within the presented framework, or as straightforward extensions, but we also foresee a possible need for alternative modelling frameworks to capture requirements of other areas of application. The framework presented here was developed as a means of exploring team modelling within the JACK paradigm, and to suggest a solution providing an efficient balance between theory and practical software engineering.

References

Cavedon, L., Rao, A., Sonenberg, L. & Tidhar, G. (1997) 'Teamwork via team plans in intelligent autonomous agent systems', *Proceedings of the International Conference on WorldWide Computing and its Applications*, volume 1274 of *LNCS*, pp 106—121.

Cohen, P.R. & Levesque, H. (1991) 'Teamwork', *Special Issue on Cognitive Science and Artificial Intelligence*, 25(4):487—512.

Cohen, P.R., Cheyer, A., Wang, M. & Baeg, S.C. (1994) 'An open agent architecture', *Proceedings of the AAAI Spring Simposium on Software Agents*, AAAI Press, pp 1-8.

Haddadi, A. (1996) 'Communication and Cooperation in Agent Systems: A Pragmatic Theory', Number 1056 in *LNCS*. Springer Verlag.

Rao, A.S. & Georgeff M.P. (1992) 'An abstract architecture for rational agents', *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR'92)* C.~Rich, W.~Swartout & B.~Nebel, editors, Morgan Kaufmann Publishers.

Tambe, M. (1997) 'Towards flexible teamwork', *Journal of Artificial Intelligence Research*, 7:83—124.

Tidhar, G., Rao,A.S, & Sonenberg, L. (1998) 'On teamwork and common knowledge' *Proceedings of the 1998 International Conference on Multi-Agent Systems ICMAS98* Paris Y.~Demazeau editor, pp 301—308.