Implementing Collective Obligations in Human-Agent Teams using KAoS Policies

Jurriaan van Diggelen[1], Jeffrey M. Bradshaw[2], Matthew Johnson[2], Andrzej Uszok[2], Paul Feltovich[2]

[1]Institute of Information and Computing Sciences
Utrecht University, the Netherlands
[2]Florida Institute for Human and Machine Cognition (IHMC),
40 S. Alcaniz, Pensacola, FL 32502, USA

jurriaan@cs.uu.nl; {jbradshaw,mjohnson,auszok,pfeltovich}@ihmc.us

Abstract. Obligations can apply to individuals, either severally or collectively. When applied severally, each individual or member of a team is independently responsible to fulfill the obligation. When applied collectively, it is the group as a whole that becomes responsible, with individual members sharing the obligation. In this paper, we present several variations of teamwork models involving the performance of collective obligations. Some of these rely heavily on a leader to ensure effective teamwork, whereas others leave much room for member autonomy. We strongly focus on the implementation of such models. We demonstrate how KAoS policies can be used to establish desired forms of cooperation through regulation of agent behavior. Some of these policies concern invariant aspects of teamwork, such as how to behave when a leader is present, how to ensure that actions are properly coordinated, and how to delegate actions. Other policies can be enabled or disabled to regulate the degree of autonomy of the team members. We have implemented a prototype of a Mars-mission scenario that demonstrates varying results when applied across these different teamwork models.

Keywords: Human-agent teams, Policies, Collective Obligations

1. Introduction

Autonomy is perhaps the most fundamental property of an agent. Generally speaking, we might say that the more control an agent has over its own actions and internal state, the greater its autonomy. By this definition, collaboration almost always entails a reduction in autonomy. In collaboration, we are willing to give up some degree of autonomy in the service of achieving joint objectives [15].

Obligations can be either voluntarily adopted or imposed. Researchers who study *norms* generally focus on the ways in which agents learn, recognize, and adopt such obligations through their own deliberation, including the consideration of incentives and sanctions [5]. Our research interest has been to understand similar issues with

respect to *policies*, constraints that are imposed and enforced prescriptively on agents [2]. Constraining an agent's collaborative activities in this way is often accomplished by virtue of the organizations to which it belongs [7][13]. The purpose of this paper is to report on the latest developments within the KAoS policy and services framework, in particular w.r.t. teamwork and collective obligations.

A KAoS policy is defined as "an enforceable, well-specified constraint on the performance of a machine-executable action by a subject in a given situation" [2]. There are two main types of polices; authorizations and obligations. Authorization policies specify which actions are permitted (*positive authorizations*) or forbidden (*negative authorizations*) in a given situation. Obligation policies specify which actions are required (*positive obligations*) or waived (*negative obligations*) in a given situation. KAoS uses OWL (Web Ontology Language: http://www.w3.org/2004/OWL) to represent policies.

KAoS policies have already been successfully applied to important aspects of joint activity in the context of human-robot teamwork [11]. In this paper, we extend this research by adding the notion of a *collective obligation* [4]. The difference between an individual obligation (IO) and a collective obligation (CO) is that in IO's each individual or member of a team is independently responsible to fulfill the obligation. On the other hand, in CO's, it is the group as a whole that becomes responsible, with individual members sharing the obligation. CO's are especially useful in governing complex abstract behavior—in our case, for example, the obligation that agents have to ensure safety. The difficulty of writing individual obligations for *ensure-safety* is that it is probably not an action that can be directly executed by any one agent. Most likely, a plan must be created to decompose *ensure-safety* into more concrete actions. It is also difficult to decide, beforehand, who is the best candidate to carry out the plan, as a different plan might be adopted in different circumstances. Moreover, agents may have different capabilities, enabling them to contribute individually or jointly in particular roles. For such reasons, constraints requiring the performance of abstract team actions like *ensure-safety* are usually better implemented as collective, as opposed to individual, obligations.

Because a CO often does not direct activity at the level of the single agent's behavior, we must find a way to translate the CO to the individual level. Our research aim in this paper can thus be described: to develop general policies to fulfill collective obligations, and to map these obligations to individuals based on the current context.

Inspired by previous theoretical groundwork on these issues [4][12], we follow a very practical approach. First, we demonstrate how to represent and reason about collective obligations in OWL. Second, we describe three sets of KAoS policies that we defined to govern agent behavior in the execution of collective obligations. Third, we provide a configuration policy set that is used to adjust specific aspects of the teamwork model for use in a given situation. Finally, we present a prototype we have implemented to demonstrate the use of these policies in the context of a Mars mission scenario [16].

We claim several benefits for developers of agent teams. The first concerns *reusability*. Because the policies describe near-universal teamwork aspects, they are domain independent and can apply to many kinds of applications, thus saving development time. The second benefit concerns *sharedness*. Because teamwork requires maintaining common ground among the participants [15], agents benefit

when the code that generates team behavior can be shared by all agents. By introducing a shared collection of teamwork policies for the whole system, in conjunction with KAoS monitoring and enforcement capabilities, newly added agents fit easily into the team, no matter who developed them or which language they are programmed in. The policies accommodate even the most primitive agents by eliminating the requirement that each agent be capable of sophisticated deliberation in order to collaborate. Next, there is the benefit of separation of concerns. By using KAoS policies, the code that implements teamwork is cleanly segregated from the rest of the agent code. This avoids the typical clutter experienced when teamwork code is scattered in arbitrary locations among all agents. Finally, KAoS policies are very straightforward to read and understand, making them more suitable to implement this kind of behavior than generic rule languages or more low-level programming languages.

In addition to the benefits for agent developers, we also believe that this approach is more conducive to scientific progress towards the much more ambitious goal of human and machine joint activity [18][8]. Although the policies described in this paper are relatively simple and elementary, they are fundamental in human teamwork. Hence, when agents adopt important aspects of human teamwork, people may find them more predictable and understandable.

The remainder of the paper is outlined as follows. Section 2 explains the basic teamwork model. Section 3 provides an overview of the KAoS policy services framework. In Sections 4, 5 and 6, we describe how we used KAoS to implement the teamwork model: ontological aspects in Section 4; policies in Section 5; an implemented prototype with agents in a Mars-mission scenario in Section 6. Related work is discussed in Section 7, followed by conclusions in Section 8.

2. Team Design

Teamwork is a topic of great complexity and breadth. Here, our focus is only on one aspect of teamwork, i.e., collective obligations. Collective obligations require teams to perform some action whenever some event or state triggers the obligation. Performing such actions typically involves planning, delegation and coordination. The aim of team design is to ensure that this process is adequately supported. Three primary aspects of team design are pertinent to the issues discussed in this paper: leadership assumption, task allocation, and plan coordination. Each of these aspects can vary, resulting in different team behavior. Figure 1 depicts these aspects in three dimensions, where each combination of aspects represents a different kind of team.

Along the x-axis, two possibilities for leadership assumption are shown. We can appoint someone as a leader beforehand (i.e. pre-established leadership), or we can defer the choice and allow leaders to volunteer on demand (i.e. *ad hoc* leadership assumption). Whereas "pre-established" and "*ad hoc*" qualify as two extremes on the leadership assumption dimension, there are, of course, intermediate options possible that we do not consider here. One example is that of a predefined line of succession which is used to determine leadership if all higher-ranking leaders are unavailable.

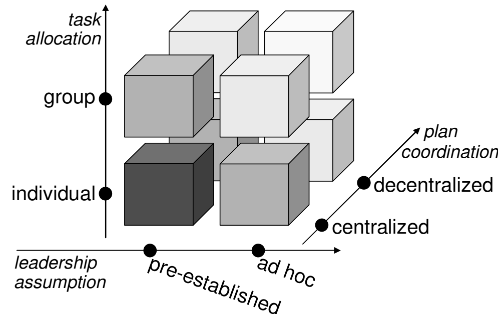

Figure 1 Three dimensions in team design

The task allocation dimension is shown along the y-axis. Individual task allocation means that requests are directed at individual agents. In group task allocation, the request is directed to the group as a whole, without specifying which individual must perform the task.

Plan coordination is depicted on the z-axis, with the two alternatives being centralized and decentralized. Figure 2 depicts the communication pattern for these two ways of coordinating plans. The left side of the figure depicts centralized coordination, i.e. the requester agent (the grey agent) is responsible for making sure that the actions are executed in the right order. The right side of the figure shows decentralized coordination, i.e. the agents executing the plan take care of the coordination themselves. In the latter case, the requester delegates plan coordination. It may do so by sending a request for action *a,* together with information about who will perform the subsequent action *b*. In the figure, this is written as "creq a,b."

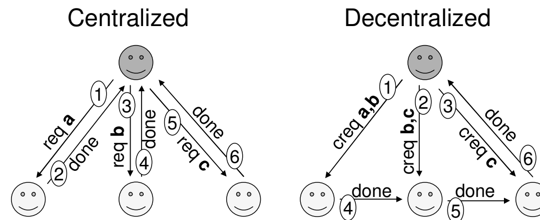

Figure 2 Centralized and decentralized coordination patterns

With centralized coordination, the requested agents may not be aware that their actions are part of a larger plan. With decentralized coordination, the requested agents require more knowledge about the action's context, i.e. they must know which agent is responsible for performing the next action in the plan.

2.1 Considerations for team design

The three dimensions outlined above can be regarded as different aspects of the dichotomy between *central authority* and *member autonomy* [3]. Pre-established leadership means that one central authority remains in charge of the team, whereas *ad hoc* leadership allows for more member autonomy because each team member may become a leader under certain circumstances. Centralized plan coordination allocates

the task of coordinating plans to one central authority, whereas decentralized plan coordination allows each agent to make its contribution to coordination, i.e. reflecting more member autonomy. Individual task allocation implies that one central authority decides who performs the tasks; whereas group task allocation yields more member autonomy as the team members decide this among themselves.

In Figure 1, the team with most central authority is represented as the black cube. For the other teams, we can say that the further away the cube is from the black cube, the more member autonomy exists in the team. The white cube represents the team with most member autonomy. Which of these eight team configurations is the best one depends on the circumstances and cannot be decided in general. Below, we outline some general considerations when choosing between central authority and member autonomy; the discussion is not intended to be exhaustive. An advantage of using a central authority might be that it allows the team designer to select the best agent for the most important tasks. In this way, the team can be better adapted to the different qualities of agents. Another advantage of a central authority approach might be accountability: that is, that it would be easier to identify the responsible agent when things go wrong.

A disadvantage of a central authority might be that it would be less robust in certain circumstances, e.g., when the leader becomes unavailable, the entire team becomes dysfunctional. Another disadvantage of central authority might arise when not every team member has the same access to the situation. For example, it may be better to have a crisis operation led by someone on site than by a predefined leader who is far away. As a last disadvantage, we mention the potentially increased response time of strongly hierarchical teams. For example, when an incident happens, this must communicated all the way up to a leader, after which the leader makes a decision and communicates it all the way down to those carrying out the work. A faster response may be obtained by allowing the observer of the incident to take immediate action.

Before we explain how these teamwork models can be implemented, we will first give some background on the KAoS policy framework.

3. KAOS POLICY FRAMEWORK

KAoS [2] provides a general framework for regulation of a variety of systems, including agent-based and robotic systems [2], web services, grid services, and traditional distributed systems. It also provides the basic services for distributed computing, including message transport and directory services, as well as more advanced features like domain and policy services.

Two important requirements for the KAoS architecture are modularity and extensibility. These requirements are supported through a framework with well-defined interfaces that can be extended, if necessary, with the components required to support application-specific policies. The basic elements of the KAoS architecture are shown in Figure 3; its three layers of functionality correspond to three different policy representations.

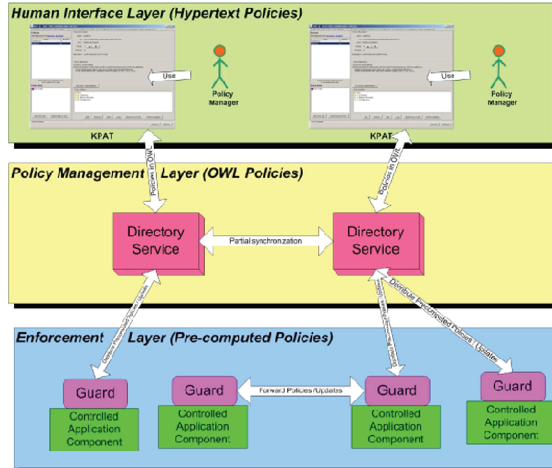

Figure 3 Notional KAoS Policy Services Architecture

- *Human Interface layer:* This layer uses a hypertext-like graphical interface for policy specification in the form of very natural English sentences, composed from pop-up menus. The vocabulary is automatically provided from the relevant ontologies, consisting of highly reusable core concepts augmented by application-specific ones.
- *Policy Management layer:* Within this layer, OWL is used to encode and manage policy-related information. The Distributed Directory Service (DDS) encapsulates a set of OWL reasoning mechanisms.
- *Policy Monitoring and Enforcement layer*: KAoS automatically "compiles" OWL policies to an efficient format that can be used for monitoring and enforcement. This representation provides the grounding for abstract ontology terms, connecting them to the instances in the runtime environment and to other policy-related information.

Maintaining consistency among these layers is handled automatically by KAoS.

3.1 System development in KAoS

Multi-agent system development in KAoS takes place at different locations, in different languages, using different tools, as summarized in the following table.

	Language	Development Tool
Agents or other Applications	E.g., Java	E.g., Eclipse
Policies	KAoS Policies (OWL)	KPAT
Ontologies	OWL	E.g., Protégé, COE

Figure 4 KAoS system development components

OWL ontologies provide the vocabulary used in specifying policies. They define all actions, action properties, and actor types and can be developed directly in OWL

or using an ontology editor, such as Protégé (http://protege.stanford.edu/) or COE (Cmap Ontology Editor).

Policies are also represented in OWL. They can be created using the KAoS Policy Administration Tool (KPAT). KPAT hides the complexity of OWL from the human users and allows the user to create, modify and manage policies in a very natural hypertext interface. Policies can be ranked in terms of their *priorities.* In case two conflicting policies are applicable at the same moment, the policy with the highest priority takes precedence.

The policies are used to govern the actions of agents (or other applications) within the system being developed. We use Java and Eclipse (http://www.eclipse.org/) to implement the agents for our prototype, although any other combination of a programming language and IDE could be used. KAoS includes a number of features that can be exploited in the development of agent-based systems.

As an example of system development in KAoS, suppose that we have a set of robots and we want to obligate them to beep before they move, in order to alert any nearby people of the pending movement. First, we would specify the terms `Robot`, `Beep` and `Move` in an ontology. Then, we would create a policy using KPAT, which would look like the following:

```
1 Robot is obligated to start performing Beep
2        which has any attributes
3 before Robot starts performing Move
4        which has any attributes
```

Figure 5 KAoS policy example

Once the policy has been created, it is sent by KPAT to the Directory Service for analysis and deconflication, before it is "compiled" and distributed to the guards for run-time enforcement. Since the policy applies only to robots, it is automatically distributed only to the guards responsible for governing robots. Local enforcement mechanisms on each platform intercept movements as appropriate and check with the guard resident on that platform for policy constraints. With the new policy in place, an obligation to beep would be applied prior to each movement.

An important part of building systems in this way is deciding where to implement a given behavior. In general, there are three possible places; in the agent, in the policies, or in the ontology (cf. Figure 4). Each has advantages and disadvantages in different situations. Without policy, we would be forced to represent everything in the agent itself, so, for our beep example, the beep action might simply be coded in Java within the move method. This is not very flexible and is hidden from those unfamiliar with the code. In situations where the source code is unavailable, it simply cannot be implemented at all. A second option is to implement the behavior by adapting the ontology, i.e. by defining a move as a beep that is followed by a physical move, and having the agents query the ontology for the definition of the action. This would amount to redefining the commonly accepted meaning of *move* into something else entirely – not be a good idea either. The third option is to add the policy of Figure 5. This seems to us the cleanest method. The policy is defined external to the robot's program and thus is viewable and editable by anyone using the system. To give an example that pushes some knowledge back into the robots, suppose that we modify our policy to state "robots must warn before they move." The main idea is still

modeled in policy, though less specific. The ontology could be used to model the knowledge that beeping and flashing lights are both appropriate methods to warn. Finally, the robot could chose the appropriate warning method based on its own capabilities and preferences.

In the following three sections we will explain how the teamwork model described in Section 3 can be implemented by developing ontologies, policies, and agents.

4. ONTOLOGY

Extending KAoS so it can handle collective obligations posed some additional requirements to the core ontology. The first issue concerned the representation of teams. The property `teamMemberOf` was used to assert that an agent (represented by an individual in class `agent`) is a member of some team (represented by an individual in class `team`). To represent the collective obligation of a team, the property `HasCollectiveObligation` was used to refer to the instance representation[1] of the action that constitutes the CO.

The second issue concerned the representation of plans. Because a plan typically consists of multiple actions, we can represent that an action contains subactions by using the properties `subAction1` and `subAction2`. The property `subActionRelation` specifies whether the two subactions are composed in parallel or sequence. In this way, composite actions can be represented as an AND-OR graph, or planning tree [17].

The last ontological issue concerned the relationship between the plan and the action the plan seeks to achieve. Because different circumstances require different plans, we specify this as a context-dependent relation, using a rule of the form "X *counts-as* Y in context C." These so-called *counts-as* rules can be used in an ontology to translate between actions of different levels of abstraction [11]. For example, the sequence of actions *bring-to-habitat* and *nurse* (the plan) counts as *ensure-safety* (what the plan is designed to achieve) in the context of *spacesuit-failure-of-Benny-at-11:00am* (the context). An action and its associated context are related by the property `hasContext`. To represent the fact that an action has been performed, the property `hasStatus` is set to `performed`. Because we represent *counts-as* rules as subclass relations (e.g. "X subClassOf Y" represents the fact that X counts as Y), the OWL reasoner automatically derives that if X `hasStatus performed`, then also Y `hasStatus performed`.

The issues discussed above are important when monitoring policy compliance. An agent complies with an obligation to do action X, if X has the status performed before the deadline set by the obligation. This definition has two important consequences. First, the agent to which the obligation applies is not required to perform the action itself, but may also delegate the action to another agent. Second, the agent can choose to perform a plan which counts as action X (in the current context), because performance of the plan entails performance of X. Both of these two issues play a fundamental role in our approach to teamwork and are therefore implemented at the ontology level.

---

[1] Because OWL-DL does not allow the use of classes as property values, we created a prototypical instance for every action class (e.g. `ensureSafety`). This prototypical instance represents the same (e.g. `ensureSafetyPrototypicalInstance`). In this way, we can refer to actions both at the class level, and at the instance level.

```
1 Leader is obligated to start performing Action which has attributes:
2  all prototypicalInstance values equal the Trigger action's
3    triggerOfCollectiveObligation of the prototypicalInstance values
4  the performedBy value equals the Trigger action's performedBy values
5 after Leader finishes performing Action which has attributes:
6  any prototypicalInstance values are in the set of this action's
7    HasCollectiveObligationTrigger of the teamMemberOf of the
8    performedBy values
```

Figure 6 KAoS hypertext statement representing the policy of Definition 1.1

5. POLICIES FOR AGENT TEAMS

The general pattern of the teamwork described in this paper consists of three steps. First, the collective obligation is triggered. Second, a plan is created. Third, this plan is carried out. The policies described in this section serve to support this process by governing issues such as: how is the CO-trigger communicated to the agent creating the plan? Who creates the plan? Who carries out the plan? How is the plan coordinated to ensure the right order of actions?

5.1 Leader Policy Set

If there is a team leader, it has a special responsibility and must be treated by the other agents in a distinct way. The purpose of the Leader Policy Set is to lay down these responsibilities, managing both task allocation and plan coordination.

Definition 1 Leader Policy Set

1. *The leader of a team should adopt the collective obligations of its team as its own individual obligations*
2. *Team members should notify their leader when the collective obligation of their team is triggered*
3. *The leader of a team may request members of its team to perform actions*
4. *The leader of a team may create plans*

The first policy captures the intuition that leaders must take responsibility for their team. Definition 1.1 states more precisely what this means for collective obligations. The policy as implemented in KAoS is shown in Figure 6. The trigger of the policy is implemented at line 5,6,7 and 8 using a role-value map [1] which compares the values of two properties of the `Action` which the agent has just finished performing. It states that the property `prototypicalInstance` must have a value in common with the concatenation of the properties `performedBy`, `teamMemberOf` and `HasCollectiveObligationTrigger`. As an example of an action that would trigger the obligation, consider agent `Herman` performing the action `observeSpaceSuitFailure` (i.e. `observeSpaceSuitFailure performedBy Herman`) and that `Herman` is `teamMemberOf MecaTeam` and that `MecaTeam HasCollectiveObligationTrigger observeSpaceSuitFailurePrototypicalInstance`. The obligation is described in lines 1, 2, 3 and 4 of Figure 6. Lines 2-3 is a role-value map which describes that the actor must do the action which is given by the property `triggerOfCollectiveObligation` of the action that triggered the obligation. In our example, `observeSpaceSuitFailure` is `triggerOfCollectiveObligation` of

`ensureSafety`. Hence, the actor is obliged to perform `ensureSafety`. Line 4 describes that the agent that must fulfill the obligation is the same agent that has triggered the obligation.

The second policy of Definition 1 ensures that, in case nobody else in the team triggers the collective obligation (for example by observing a spacesuit failure), this agent will notify the leader about the event. This captures the intuition that team members must help their leader. This policy is implemented in a similar fashion to policy 1.1 (Figure 6).

The third policy in the leader policy set states that leaders do not have to do the work all by themselves, but they are authorized to request actions from their team members.

The fourth policy states that the leader is authorized to create a plan. Plan creation is done by adding a *counts-as* rule to the ontology (see Section 4). The effect of this is that all agents may perform a different action than the action they were initially obliged to do. Therefore, the right to create new plans is not self-evident. It is, however, a right that belongs to a leader.

5.2 Coordination Policy Set

The coordination policy set describes how actions in a plan should be coordinated. We consider two coordination patterns (as depicted in Figure 2), which are both governed by this policy set.

Definition 2 Coordination Policy Set

1. *An agent should notify the requester after it has performed a requested action*
2. *If the agent knows who will perform the subsequent action, it should notify that agent after it finishes performing its own action*
3. *If the agent knows who will conduct the subsequent action, it is not required to notify the requester after it finishes performing its action*

The first policy ensures that, in case of centralized coordination, the requester knows when the subsequent action may begin. This is due to the “done” messages 2, 4 and 6 on the left side of Figure 2. In case of decentralized coordination, the requester is notified after the plan is finished, i.e. by “done” message 6 on the right side of the figure.

The second policy of Definition 2 concerns the case of decentralized coordination. When an agent has received a request for a coordinated action, it knows who will perform the subsequent action, and must notify that agent after it has finished its action.

The third policy is enforced with high priority, and can be regarded as an exception to the first policy of Definition 2. This policy prevents requested agents from notifying their requester when the plan is only partially completed. As can be seen on the right hand side of Figure 2, the two agents that are requested to perform action *a* and action *b* of the plan do not send a “done” message to their requester. The rationale behind this is that, in the decentralized case, partially-finished notifications are not needed for *plan coordination*, which is the purpose of this policy set. There may be other reasons why this may be desirable, e.g., to monitor plan progress to respond to unexpected events in a timely way [8]. This can always be implemented in

an additional higher priority policy set, which is specially designed for that purpose. However, issues such as dealing with plan failure or replanning are issues of future research.

5.3 Leader Absence Policy Set

What if the agents find themselves in a leaderless team? This may happen either because nobody has been appointed as a leader or else the leader is (temporarily) unavailable. In this case, the other agents in the team must take care of the collective obligation themselves. This issue is handled by ensuring that one agent assumes the leader role, and thereby becomes subject to the leadership policies of Definition 1.

Definition 3 Leader Absence Policy Set

1. *When no leader is present, the CO is triggered, and the agent knows it can fulfill the CO, it should assume the leader role*
2. *When no leader is present, the CO is triggered, but the agent cannot fulfill the CO, it should notify the whole team of the CO trigger*
3. *An agent should not notify its team about a CO trigger, when it has been notified itself by another team member about that CO trigger*

The first policy ensures that a capable leader will volunteer in case the collective obligation is triggered in a leaderless team. An agent may assume leadership by registering with the KAoS directory-service, which only accepts such a registration when there are no other leaders already currently available. In this way, we prevent multiple agents from taking leadership at the same time, on a first come, first served basis.

The second policy is a variation on the policy of Definition 1.2, adapted to the leaderless scenario. For example, when an agent observes a safety critical event (the CO is triggered), but the agent is not capable of ensuring safety, the agent should notify all of its team members about it, so someone else in the team can fulfill the CO.

The third policy is an exception to the second rule, and prevents agents from repeatedly notifying one another about the same collective obligation trigger.

5.4 Configuration Policy Set

The policies discussed so far are the same for all eight different kinds of teams depicted in Figure 1. In this section, we will discuss the configuration policy set which states which of the eight team strategies the agents must follow.

Definition 4 Configuration Policy Set

1. *Do not request distributed coordinated actions*
2. *Do not request actions to a team*

In contrast to the policy sets we discussed earlier, these policies are optional, and can be switched on and off depending on the way the team designer wishes to configure the team. If the first policy is switched on, the team will apply centralized plan coordination. If it is switched off, the team will apply decentralized plan coordination.

If the second policy is switched on, the team will apply individual task allocation. If it is switched off, the team applies group task allocation. Group task allocation can be implemented using collective obligations that are dealt with using the policies described in the previous sections. For example, to request action *a* to a group, the action *a* is added as a collective obligation to that group. The leader absence policy set (Definition 3) ensures that a leader which is capable of performing action *a* stands up, after which the leader policy set (Definition 1) ensures that this agent performs action *a*.

To implement pre-established leadership assumption, a leader must be appointed beforehand, using KPAT. To implement *ad hoc* leadership assumption, no leader should be defined beforehand, such that the policy in Definition 3.1 ensures that a leader will volunteer at runtime if needed.

6. MECA SCENARIO

We tested the policies using a Mars mission scenario developed in the Mission Execution Crew Assistant (MECA) project [16]. This long-term project aims at enhancing the cognitive capacities of human-machine teams during planetary exploration missions by means of an electronic partner. The e-partner helps the crew to assess a situation and determine a suitable course of actions when problems arise. A large part of the project is devoted to developing a requirements baseline, taking into account human factors knowledge, operational demands, and envisioned technology. Developing new prototypes using emerging technologies, such as this one, is a continuous activity in the project.

One of the major themes is dealing with the long communication delays between Earth and Mars. This has led researchers to consider new forms of mission control that are less centralized on Earth, allowing greater autonomy to the astronauts on Mars [10]. We believe that our work on policies and team strategies is a useful contribution to this problem.

One of the use-cases that has driven the development of MECA's requirements baseline concerns an astronaut suffering from hypothermia. The initial situation is depicted in Figure 7.

Herman is in the Habitat; Anne, Albert and two rovers are in team A; Benny and Brenda are in team B. Benny and Brenda are on a rock-collecting procedure. Suddenly, Benny's space suit fails. Brenda and the MECA system diagnose the problem together and predict hypothermia. Immediate action is required. A rover from team B comes to pick Benny up and brings him to the habitat. Someone with surgery skills and someone with nursing skills await him there and take care of Benny, after which he safely recovers.

One of the requirements of MECA is that safety of the crew must be ensured at all times. We implemented this requirement using a collective obligation of the MECA team to *EnsureSafety*. The trigger of this collective obligation is *Observe-SafetyCriticalEvent.* Within the scenario, both of these actions are added in a specific MECA-action ontology which extends the KAoS core action ontology. The ontology also specifies several subconcepts of *ObserveSafetyCriticalEvent*, such as *Observe-SpaceSuitFails.* This causes *ObserveSpaceSuitFails* to trigger the collective obligation.

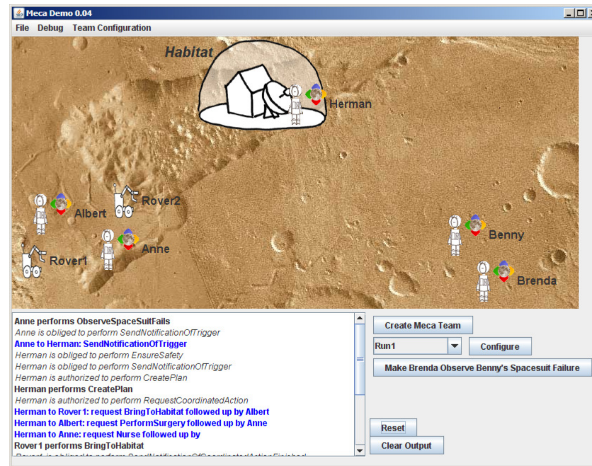

Figure 7 MECA prototype

The seven agents in the example (five astronauts and two rovers) are implemented in Java. Because most of the agent behavior in this demonstration is implemented by the policies, the Java implementation could remain very simple. We used Java to implement how the actions, such as *BringToHabitat*, are performed. For the purposes of this demonstration, a simple screen animation was sufficient. We also implemented in Java how the agents remain policy-compliant. This means that they consult the KAoS guard to check which obligations and authorization policies apply. They fulfill an obligation by simply executing the code that implements the action concerned. It fulfills a negative authorization by refraining from executing the corresponding piece of code.

The most important aspect of this demonstration is the unfolding of the scenario after the action *ObserveSpaceSuitFails* is performed. This is driven exclusively by KAoS policies. By applying the different team configurations described in Section 0, we obtain different event traces which demonstrate the functioning of the team. The event trace for the most centrally organized team (represented by the black cube in Figure 1) is shown below.

Brenda performs ObserveSpaceSuitFails
Brenda is obliged to perform SendNotificationOfTrigger
Brenda to Herman: SendNotificationOfTrigger
Herman is obliged to perform EnsureSafety
Herman is authorized to perform CreatePlan
Herman performs CreatePlan
Herman is not authorized to perform RequestCoordinatedAction
Herman is authorized to perform RequestAction
Herman to Rover1: request BringToHabitat
Rover1 performs BringToHabitat
Rover1 is obliged to perform SendNotificationOfRequestedActionFinished
Rover1 to Herman: SendNotificationOfRequestedActionFinished
Herman to Albert: request PerformSurgery
Albert performs PerformSurgery
Albert is obliged to perform SendNotificationOfRequestedActionFinished
Albert to Herman: SendNotificationOfRequestedActionFinished
Herman to Anne: request Nurse

Anne performs Nurse
Anne is obliged to perform SendNotificationOfRequestedActionFinished
Anne to Herman: SendNotificationOfRequestedActionFinished

Figure 8 Event trace of MECA team with maximal central authority

The events printed in bold are actions; the underlined events are communication actions; the italicized events represent policies that were triggered. Typical to this event trace is that Brenda immediately knows that she must contact Herman after she observed the spacesuit failure. This is due to the pre-established leadership of Herman. Furthermore, Herman delegates the parts of the plan to individual agents (i.e. individual task allocation), and he waits until the requested agent is finished before he requests the next action in the plan (i.e. centralized plan coordination).

The event trace for the team with most member autonomy (represented by the white cube in Figure 1) is shown in Figure 9.

Brenda performs ObserveSpaceSuitFails
Brenda is obliged to perform SendNotificationOfTrigger
Brenda to Rover1: SendNotificationOfTrigger
Brenda to Anne: SendNotificationOfTrigger
Brenda to Albert: SendNotificationOfTrigger
Brenda to Rover2: SendNotificationOfTrigger
Brenda to Herman: SendNotificationOfTrigger
Brenda to Benny: SendNotificationOfTrigger
Anne is obliged to perform AssumeLeaderRole
Anne is obliged to perform EnsureSafety
Anne is authorized to perform CreatePlan
Anne performs CreatePlan
Anne is authorized to perform RequestCoordinatedAction
Anne is authorized to perform TeamRequestAction
Anne to MecaTeam: request BringToHabitat
Anne to MecaTeam: request PerformSurgery after BringToHabitat
Anne to MecaTeam: request Nurse after PerformSurgery
Rover1 is obliged to perform AssumeLeaderRole
Rover1 performs BringToHabitat
Rover1 is obliged to perform SendNotificationOfTrigger
Rover1 to MecaTeam: SendNotificationOfTrigger
Albert is obliged to perform AssumeLeaderRole
Albert performs PerformSurgery
Albert is obliged to perform SendNotificationOfTrigger
Albert to MecaTeam: SendNotificationOfTrigger
Herman is obliged to perform AssumeLeaderRole
Herman performs Nurse

Figure 9 Event trace of MECA team with maximal member autonomy

Typical to this event trace is that Brenda notifies the whole team about the CO trigger, after which Anne becomes a leader (i.e. *ad hoc* leadership assumption). Furthermore, Anne delegates her actions to the MECA team (i.e., group task allocation). Also, she delegates all actions at once and instructs the agents how to coordinate the actions (i.e., decentralized plan coordination).

7. RELATED WORK

A similar approach to teamwork, based on electronic institutions, is reported in [9]. This framework captures coordination aspects by dynamically composing existing teamwork components, s.a. communication protocols and operational descriptions, to meet the current problem requirements. Our approach is more centered around the

idea of constraining autonomy, i.e. by using computational policies as basic teamwork components.

The pioneering research of Cohen and Levesque [4] introduced the notion of a *joint persistent goal* as the ultimate driving force behind teamwork. In our framework, a collective obligation serves a similar purpose. A difference is that Cohen and Levesque based their approach on mentalistic notions, such as goals, beliefs and intentions, whereas our approach is based on institutional notions, such as obligations and authorizations. This allows the approach to be used by both simple and sophisticated agents, of heterogeneous varieties.

A similar difference can be observed when comparing our implementation with other teamwork model implementations, such as STEAM [1]. STEAM is based on Soar, a general cognitive architecture for intelligent systems, whereas our approach is based on KAoS, which is a policy framework. A correspondence between our implementation and STEAM is that both approaches heavily rely on plans in the teamwork process. A crucial requirement for effective teamwork is maintaining a sufficient level of common ground [15]. By adopting the KAoS framework, some important aspects of common ground were naturally ensured. The common ontology, which is maintained by the directory service and distributed to the guards, ensures that every agent shares understanding of the domain terms. Also the collective obligations of the team, which are represented in the ontology, are mutually known.

8. CONCLUSION

In this paper, we have proposed a policy-based approach for human-agent teams. We have implemented a variety of teamwork models in KAoS. These models have demonstrated their value in a simulation of a Mars-mission scenario, where a delicate decision must be made between central authority and member autonomy.

We believe that our approach to teamwork has considerable benefits in terms of reusability, clarity, and generality. Although the types of teamwork we support are still elementary, we believe that more complex teamwork can be implemented by utilizing additional policies on top of the policies we have proposed here.

In the future, we plan to extend the teamwork model to deal with unexpected events. This requires a leader to monitor his or her plan, and to perform replanning if the plan does not go as expected. Also, the team members can be of help here by notifying their leaders when their requested actions fail (cf. [8]). Such policies can be implemented in KAoS, in a similar fashion as we have described in this paper.

REFERENCES

[1] Baader, F., Calvanese D., McGuinness, D. L., Nardi D., and Patel-Schneider, P. F. Eds. (2003). The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press.

[2] Bradshaw, J. M., et al. (2003). Representation and reasoning for DAML-based policy and domain services in KAoS and Nomads. Proceedings of the Autonomous Agents and Multi-Agent Systems Conference (AAMAS), ACM.

[3] R.M. Burton, G. DeSanctis, B. Obel (2006), Organizational Design, Cambridge University Press.

[4] Cohen, P.R. and H.J. Levesque. (1991). Teamwork. Menlo Park,CA: SRI International.

[5] Davidsson, P. (2000): Emergent Societies of Information Agents. Klusch, M, Kerschberg, L. (Eds.): *Cooperative Information Agents IV*, LNAI 1860, Springer, 2000, pp. 143–153.
[6] Dignum, F. and Royakkers, L. (1998). Collective Obligation and Commitment, In Proceedings of 5th Int. conference on Law in the Information Society, Florence
[7] Dignum, V. (2003). A Model for Organizational Interaction. SIKS Dissertation Series.
[8] Feltovich, P.J., Bradshaw, J.M., Clancey, W.J., Johnson, M., & Bunch, L (2008). Progress appraisal as a challenging element of coordination in human and machine joint activity. In Engineering Societies in the Agents' World VIII. Lecture Notes in Computer Science Series. Heidelberg Germany: Springer.
[9] Gómez, Mario; Plaza, Enric (2008). Dynamic Composition of Electronic Institutions for Teamwork.Coordination, Organizations, Institutions, and Norms in Agent Systems III. (COIN)., LNAI, Vol. 4870, pp. 155-170. Springer Verlag.
[10] Grant, T., Soler, A. O., Bos, A., Brauer, U., Neerincx, M., and Wolff, M. 2006. Space Autonomy as Migration of Functionality: The Mars Case. In *Proceedings of the 2nd IEEE international Conference on Space Mission Challenges For information Technology* (SMC-IT). IEEE, 195-201.
[11] Grossi, D. (2007). Designing Invisible Handcuffs. Formal Investigations in Institutions and Organizations for Multi-agent Systems. SIKS Dissertation Series 2007-16, Utrecht University.
[12] Grossi, D., Dignum, F., Royakkers L., Meyer, J-J. Ch., (2004). Collective Obligations and Agents: Who Gets the Blame? Proc. of DEON'04, 7th Int. Workshop on Deontic Logic in Computer Science. Springer. LNCS 3065
[13] Hübner, J. F., Sichman, J. S., and Boissier, O., (2002). A Model for the Structural, Functional, and Deontic Specification of Organizations in Multiagent Systems. In Proc. of the 16th Brazilian Symposium on AI, LNCS, vol. 2507. Springer-Verlag, London, 118-128.
[14] Johnson, M. J., Intlekofer, K., Jr., Jung, H., Bradshaw, J. M., Allen, J. Suri, N. & Carvalho, M., (2008). Coordinated operations in mixed teams of humans and robots. In Proceedings of the 2008 IEEE International Conference on Distributed Human-Machine Systems (DHMS 2008)., pp. 63-68.
[15] Klein, G. Woods, D., Bradshaw, J.M. Hoffman, R.R. , Feltovich, P.J. (2004). Ten Challenges for Making Automation a Team Player. In Joint Human-Agent Activity," IEEE Intelligent Systems, vol. 19, no. 6
[16] Neerincx, M.A. Bos, A., Olmedo-Soler, A. Brauer, U. Breebaart, L., Smets, N., Lindenberg, J., Grant, T., Wolff, M. (2008). The Mission Execution Crew Assistant: Improving Human-Machine Team Resilience for Long Duration Missions. Proc. of the 59th International Astronautical Congress (IAC2008)
[17] Steffik, M. (1995). Introduction to Knowledge Systems, Morgan Kaufmann Publishers.
[18] Sycara, K., and Lewis, M. 2004. Integrating intelligent agents into human teams. In Team Cognition: Understanding the Factors that Drive Process and Performance,, 203-232. Washington, DC: American Psychological Association.
[19] Tambe, M. (1997). Towards Flexible Teamwork, Journal of Artificial Intelligence Research, pp. 83-124