Team Oriented Programming and Proxy Agents: The Next Generation

Paul Scerri*, David V. Pynadath+, Nathan Schurr+, Alessandro Farinelli+, Sudeep Gandhe+ and Milind Tambe+

* Carnegie Mellon University
+ University of Southern California
pscerri@cs.cmu.edu, pynadath@isi.edu ,{schurr,farinelli,gandhe,tambe}@usc.edu

Abstract. Coordination between large teams of highly heterogeneous entities will change the way complex goals are pursued in real world environments. One approach to achieving the required coordination in such teams is to give each team member a *proxy* that assumes routine coordination activities on behalf of its team member. Despite that approach's success, as we attempt to apply this first generation of proxy architecture to larger teams in more challenging environments, some limitations become clear. In this paper, we present initial efforts on the next generation of proxy architecture and Team Oriented Programming (TOP), called Machinetta. Machinetta aims to overcome the limitations of the previous generation of proxies and allow effective coordination between very large teams of highly heterogeneous agents. We describe the principles underlying the design of the Machinetta proxies and present initial results from two domains.

1 Introduction

Exciting emerging applications require hundreds or thousands of robots, agents and people (RAPs) to coordinate to achieve their joint goals. In domains such as military operations, space or disaster response, coordination between large numbers of agents can revolutionize our ability to achieve complex goals. Such domains are characterized by widely distributed entities with limited communication channels between them. The environments often change dynamically and will in some cases be hostile.

An emerging standard for creating robust, highly heterogeneous teams is an architecture that uses semi-autonomous *proxy agents* to create an homogeneous coordination layer "above" the highly heterogeous agents [18]. By providing each agent with a proxy that has teamwork knowledge we raise the coordination ability of the agent-proxy pair to the level at which the rest of the team is operating. Via the use of adjustable autonomy [9], the amount of coordination ability that proxy actually provides (and the amount which the agent provides) is dynamically adjusted to the particular context of the agent at that point in time. The proxies manage the coordination, performing the routine operations that are required for cooperation; e.g., informing others when plans are completed, and assisting in the handling of exceptional situations; e.g., finding RAPs to fulfill roles due to failures or overloading. The proxies also assist in making adjustments to the plan the group is following when required. The proxies also perform more routine tasks, such as information sharing, that ensure the continued smooth operation of the team, while freeing the agent from engaging in these activities. One approach to achieving the required coordination in teams for such domains, is to give each team member a *proxy* that assumes routine coordination activities on behalf of its team member[11].

The relatively homogeneous proxies allow developers to write *Team Oriented Programs* (TOPs) which are executed according to the coordination algorithms the programmer knows the proxy is using[12]. For example, TEAMCORE proxies implement the STEAM interpretation of teamwork[17]. Programmers creating TOPs need not concern themselves with specifying low level coordination details. Instead, they specify the activities of the team with high level primitives such as *roles* and *team plan operators*. Writing TOPs at this high level of abstraction makes it feasible for a programmer to quickly specify complex team activities. A variety of interesting applications have shown the utility of the proxy based teams architecture, e.g., the Electric Elves[1] and interactions with autonomous systems for space missions[15].

Despite their successes, the first generation proxy architectures suffer from three key limitations when handling: scale, dynamism, and effective integration of humans in agent teamwork. First, in small-scale teams, agents can be allocated to roles by hand prior to starting up team activities; although limited reallocation can occur at run-time. Unfortunately, in large-scale teams, off-line allocation of agents to roles by hand is difficult. Second, a high level of dynamism in the environment requires that agents' role allocation and reallocation strategy must often be integrated into one unified fluid algorithm, rather than as two separate phases (allocation and reallocation) as seen in existing architectures. Furthermore, agents must consider role reallocation not only under catastrophic failures (as was done previously), but must be willing to give up current roles to take up new precious opportunities. Third, as we build increasingly heterogeneous teams, and particularly include humans in the loop, we must enable the proxies to tap into human expertise when coordinating key situations. Previous research on teamwork has allowed agents and humans to work together, but the human participation was limited to domain level activities. Proxies should be able to use human enterprise for both domain activities and coordination.

In this chapter, we present initial efforts on the next generation of TOP and proxy architecture, called Machinetta. Machinetta aims to overcome the limitations of the previous generation of proxy and allow effective coordination between very large teams of highly heterogeneous agents. To achieve this, Machinetta embodies several new design principles. To address the first two limitations discussed above, Machinetta has a fluid, integrated role allocation and reallocation algorithm. Within this algorithm, agents attempt to continually allocate and reallocate themselves to new tasks. When new tasks/opportunities arise, or when agents' capabilities decline substantially, agents reconsider their current commitments to roles; thus, agents may change roles even without catastrophic failures. This new integrated algorithm enables a much more flexible response to dynamic environments.

Many heuristics used by a team fail under some circumstances. However, the answer is not simply to replace current coordination algorithms with new ones. If a big enough team is put into a complex enough environment there are, despite our best efforts, bound to be situations where any coordination algorithms perform very poorly or breaks altogether. A key idea in Machinetta is to acknowledge that such problems are going to occur and build in mechanisms for meta-reasoning to handle those situations. This is achieved by making as much of the coordination process as possible explicit, thus making it easier to monitor the coordination and understand when problems occur. For example, we use a role allocation algorithm[14] that represents each role to be allocated as an explicit role. If the role of allocating the role goes unachieved for some period of time, i.e., because the standard role allocation algorithm does not succeed in allocating it, the team can detect this situation and recursively invoke meta-reasoning about a "role allocation role".

With respect to involving humans in coordination (and not just in domain-level tasks), the meta-reasoning capability provides a helpful mechanism. In particular, when meta-reasoning about coordination, agents can appeal to human input. However, humans could provide input that may not necessarily be in agreement with choices made by the coordination algorithm. Thus, given the possibility of such arbitrary changes by humans to coordination algorithms, the algorithms must be robust to decisions that are "wrong" according the algorithm. For example, a human may arbitrarily (so far as the proxies are concerned) decide to terminate a plan and the proxies must implement this decision.

The final change in direction for the new generation of proxy is the properties that we aim to prove for the key algorithms. With relatively small teams, establishing properties such as optimality is important. However, proofs of such algorithm properties typically rely on assumptions such as the underlying situation not changing while the algorithm is executing. While such assumptions are very reasonable for small teams, they are not so interesting for very large teams where the assumptions will never be met. The critical point is that large enough teams in complex enough environemts will be in a constant state of change. For example, in a large team for disaster recovery in a large city, some team member will always be completing, abandoning or beginning a task. The inherent, continuous dynamics makes other algorithmic properties interesting. For example, the "stability" of the system – will one team member's failure to complete a role lead to many role reallocations or will the effects be limited? Another interesting property would be to show that certain events will never happen, or happen only with a very low probability. For example, we may be able to prove that some team member will always eventually accept a role, if its priority is above some threshold. Our current approach is to use the theory of dynamic patterns [7] to model and understand the system's properties.

In the remainder of this chapter we present Machinetta in detail, showing how it embodies the principles discussed above. We also present a graphical development environment for specifying Machinetta plans. We show preliminary results from using Machinetta in two domains, a fire fighting domain and a distributed sensor domain.

2 Proxies

Machinetta proxies are lightweight, domain-independent software modules, capable of performing the activities required to work cooperatively within a larger team on TOPs. The proxies are implemented in Java and are designed to run on a number of platforms including laptops and handheld devices. A proxy's software is made up of five components (see Figure 1):

Communication: communication with other proxies
Coordination: reasoning about team plans and communication
State: the working memory of the proxy
Adjustable Autonomy: reasoning about whether to act autonomously or pass control to the RAP
RAP Interface: communication with the RAP

Each component abstracts away details allowing other components to work without considering those details. For example, the RAP interface component is aware of what type of RAP it is connected to and the methods of interacting with the RAP, while the adjustable autonomy component deals with the RAP as an abstract entity having particular capabilities. Likewise, the communication component will be tailored to the RAP communication abilities, e.g., wireless or wired, but the coordination component will only be told available bandwidth and cost of communication.

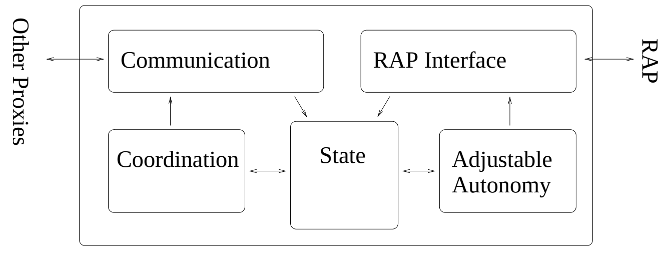


Fig. 1. Proxy software architecture.

A critical component in deploying the proxies is the mechanism by which they interact with their RAPs. The adjustable autonomy component is responsible for deciding what interaction should happen with the RAP, but the RAP interface component manages that interaction. The RAP interface component is the only part of the proxy that needs to be designed for a specific type of RAP. These components are very diverse, matching the very diverse RAPs. For example, the RAP interface for a person playing the role of fire chief in the disaster rescue domain is a large graphical interface, while for the agents, a simple socket communicating a small, fixed set of messages is sufficient. Since the proxies interact closely with their RAPs, it is desirable to have them in close physical proximity. For mobile RAPs, the proxies can be run on handheld devices that communicate wirelessly with robots or, in the case of a person in the field, via a graphical interface on the handheld device.

2.1 Proxy Algorithms

The proxy's overall execution is message driven. When a message comes in from its RAP or from another proxy, a new *belief* is added to the proxy's *state*. The beliefs in the state constitute the proxy's knowledge of the status of the team and the environment. The state operates as a blackboard, with components writing information to the blackboard and others reacting to information written to the blackboard. Any change to the state triggers two reasoning algorithms: *Coordination* (Algorithm 1) and *Adjustable Autonomy* (Algorithm 2). Either of these algorithms may in turn change the belief state, which will once again trigger the algorithms.

Algorithm 1 shows the Coordination algorithm, which instantiates the theory of *joint commitments*[3] as operationalized by STEAM. The functions, *establishJointCommitment* and *endJointCommitment* establish or terminate commitments by communicating with other proxies when a new belief triggers the start or end of a team plan. In the algorithm, the function *communicate?* returns true if the capability or role progress information should be communicated to others. This function encapsulates previous work on determining policies for communicating such information with team members[10].

Algorithm 1: Coordination

```
COORDINATION($\mathbf{B_{in}}$)
(1)  foreach $b \in \mathbf{B_{in}}$
(2)      if $b$ is CapabilityInformation or Role Progress
(3)          if communicate?($b$)
(4)              sendToOthers($b$)
(5)      else if startTeamPlan $\alpha$?($b$)
(6)          establishJointCommitment($\alpha$)
(7)          ALLOCATEROLErole($\alpha$)
(8)      else if endTeamPlan $\alpha$?($b$)
(9)          endJointCommitment($\alpha$)
(10) return $\mathbf{B_{out}}$
```

The Adjustable Autonomy algorithm (Algorithm 2) is responsible for managing the interactions between the proxy and the RAP. In the algorithm, the function *tellRAP?* determines if there is value in sending this particular piece of information received from another proxy to the RAP. The *shouldRAPbeAsked*? is the "core" of adjustable autonomy reasoning and is responsible for deciding whether or not this particular coordination decision should be handled autonomously by the proxy or by the RAP. The *if* statement beginning on Line 2 shows the basic processing that the proxy performs when its RAP is offered a new role. First, it decides whether to act autonomously. If so, it decides whether or not to accept the role on behalf of the RAP (see next section for more detail).

Algorithm 2: Adjustable Autonomy

```
AdjustableAutonomy($\mathbf{B_{in}}$)
(1)  foreach $b \in \mathbf{B_{in}}$
(2)      if $b$ is role offer
(3)          if RAP is capable of role
(4)              if shouldRAPbeAsked?
(5)                  Ask RAP
(6)              else if accept autonomously?
(7)                  $B_{out} \leftarrow$ role accepted
(8)              else
(9)                  $B_{out} \leftarrow$ role rejected
(10)     else if $b$ is a new role
(11)         send role to RAP
(12) return $\mathbf{B_{out}}$
```

3 Executing Team Oriented Plans

Within Machinetta, team plans provide an explicit representation of the joint goals held by all team members. As such, they allow the team members to scope their reasoning and concentrate on only those tasks that are directly relevant to the team's currently active goals. Due to its intended use as a domain-independent coordination architecture, Machinetta makes minimal assumptions about the nature of team plans. In other words, the team plans provided by the architecture are a skeleton of execution that the system designer then fleshes out with the intended domain-specific behavior (e.g., as part of a specific RAP behavior that triggers off team plans).

As in the original STEAM-based Teamcore architecture [11], we implement the joint goals of the TOP via reactive team plans. Active team plans take the form of beliefs within the proxies' state. This explicit representation enables the underlying architecture to reason about the means of ensuring coherent plan execution. Because each proxy maintains separate beliefs about these joint goals, the architecture can detect (in a distributed manner) any inconsistencies among team members' plan beliefs. The architecture's primary responsibility regarding coherent team beliefs about active goals is to synchronize the initiation and termination of team plans. Perhaps more importantly, the proxy must also ensure that the team makes progress toward achieving its active joint goals. The proxies themselves have no ability to achieve goals at the domain level; instead, they must ensure that all of the requisite domain-level capabilities are brought to bear by instantiating the appropriate roles and filling them with the appropriate RAPs. Section 3.1 describes the initiation of team plans, Section 3.2 describes the instantiation of the associated roles, and Section 3.3 describes the termination of team plans.

3.1 Plan Initiation

Machinetta's proxy-based infrastructure ensures that the team will synchronize itself appropriately in initiating a new team plan. Thus, the team programmer need not program such synchronization actions, because the proxies (through the *establishJointCommitment* procedure in Algorithm 1) ensure such synchronization, so all team members will agree on the set of active team plans.

The most common mechanism for creating team plans is to write a "team plan template". Such a template represents a class of possible plan instantiations. We thus save on specification effort, since writing one team plan template replaces the specification of many individual plans themselves. For example, we can write one template to represent a generic plan of "Fight a fire at building x", rather than writing hundreds of plans of the form "Fight a fire at building 1", "Fight a fire at building 2", "Fight a fire at building 3", etc.

When the preconditions of a plan template match the proxy's current state of beliefs (i.e., when $startTeamPlan\ \alpha$ is true), a new plan belief is instantiated with the specific details of the particular precondition match. This team plan belief can then trigger domain-specific behavior through the interface with a proxy's specific RAP. The proxies dynamically instantiate plans when, during the course of execution, their current states match a plan's required trigger conditions. The preconditions specify those trigger conditions, templates against which the proxies try to match active belief objects in their proxy state. Because we cannot anticipate all of the possible structures that a belief object may take on, we perform this matching by converting the belief object into some canonical string representation.

The preconditions may also include coordination constraints among team plans. For example, subgoal relationships translate into an additional precondition on child plans (e.g., if α_1 is a subgoal of α_0, then there is a precondition for α_1 requiring a current plan belief α_0). We can also specify temporal constraints between parallel subgoals (e.g., if α_1 must complete before α_2 begins, then there is a precondition for α_2 requiring a current plan belief α_1 that has been completed). Thus, the architecture can automatically translate coordination constraints specified at the abstract plan level into specific preconditions at the coordination policy level.

Upon successfully triggering a new plan, the proxies perform the *establishJointCommitment* procedure specified by their coordination policy. For example, in the initial stages of development, we used a naive communication policy that established commitments by requiring communication of all beliefs. Because all of the proxies are truthful and because we assume perfect communication, such a policy necessarily achieves mutual belief of active team plans. We have also implemented the STEAM policy [17] as a communication policy that is able to more flexibly balance the costs and benefits of communication during the establishment of a new commitment.

3.2 Role Instantiation

Roles are slots for specialized execution that the team may potentially fill at runtime. Upon instantiation of a newly triggered plan, the proxies also instantiate any associated roles, subject to the specific triggers. The specification of such roles is domain-specific, and may include appropriate role relationships, such as *AND*, *OR*, and *role-dependency* relationships (using STEAM semantics [17]). The initial plan specification may name particular RAPs to fill these roles, but more typically, the roles are instantiated unfilled. These unfilled roles are then subject to role allocation, as specified by the AllocateRole call in Algorithm 1.

3.3 Plan Termination

As in the original Teamcore architecture [11], the TOP includes each plan's termination conditions, under which a team plan is achieved, irrelevant or unachievable. Such explicit specification ensures common knowledge of such conditions, so that the team can terminate the goal coherently. Machinetta then automatically uses the termination conditions as the basis for automatically generating the communication necessary to jointly terminate a team plan.

Postconditions are roughly identical to preconditions, except for the obvious difference that the conditions contained within a postcondition refer to plan termination rather than initiation. Furthermore, the conditions are not matched against arbitrary beliefs, but rather against only those beliefs stored within the relevant container belief object (e.g., a plan). There is another key difference, in that we differentiate among three different types of termination states. In particular, we distinguish whether a plan terminated because it has become achieved, unachievable, or irrelevant (following the STEAM semantics [17]).

When a proxy's current beliefs match the postconditions of a currently active team plan (i.e., $endTeamPlan\ \alpha$ is true), the proxy triggers the plan termination phase of Algorithm 1. Again, our developmental coordination policy simply communicated the terminated plan belief (along with the domain-specific that triggered the termination) to all of the team members. We have also implemented the STEAM policy as an alternate *endJointCommitment* procedure that is capable of communicating only selectively.

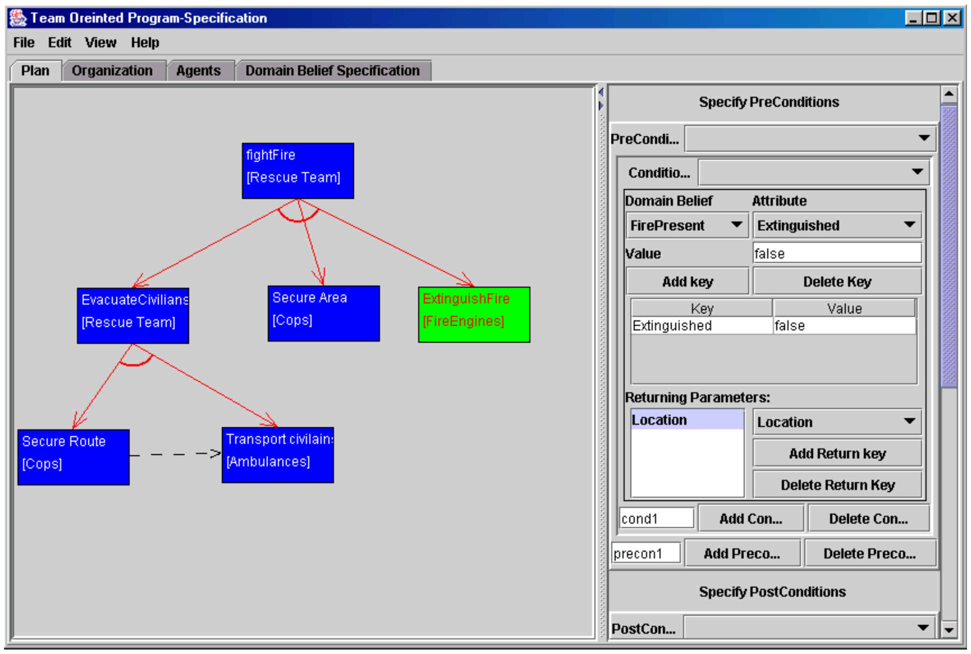
4 Specifying Team Oriented Plans

As a step towards realizing Team Oriented Programming paradigm, we have built a Java based Graphical User Interface (GUI) to facilitate domain experts to specify these TOPs. We view a TOP consisting of three parts: team organization hierarchy and descriptions of individual agents and their capabilities. These are specified using the tool in form of diagrams and are finally converted to beliefs each agent should hold to start with. These beliefs are described in XML. We have designed a XML scheme that can specify reactive plans and constraints between them, also different roles and capabilities of agents. Current Machinetta work has focused on developing proxies capable of executing simple team plans and we are in the process of extending the capabilities of the proxies to incorporate all the aforementioned features. The key thing to note about our specifications of TOPs is what is not specified. Specifically, nothing about how the coordination should be performed is explicitly specified.

Our TOP specification interface has 3 views; the first is plan hierarchy where user can draw reactive plans with plans/subplans as nodes and links showing hierarchy between them. Each subplan can have preconditions and postconditions and plan body. The agents will instantiate a team plan when preconditions match with the environment. One can specify what information will be passed to the instantiated team plan while specifying preconditions. Figure 1 shows the use of tool to specify TOP for a Robocup Rescue scenario. By using hierarchy in plans we can break down a complex plan in parts as in this case fightFire can have two subplans to evacuate civilians, secure the area, extinguish the fire. When a particular subplan's success depends on coordination with other activity we can specify coordination constraints between them. For example, such constraints can be useful in specifying that activities of transport of civilians and securing the route for transport vehicles must be coordinated. Also AND/OR constraints between subplans can be specified, for example, while fighting a fire, evacuating people, securing an area and extinguishing fire, all actions should be done, failure in any subplan can result in failure of fightFire plan, thus the designs should use an AND constraint.

A subplan can be chosen by double-clicking it and pre and post conditions for it can be specified in the right subpanel. Conditions themselves have a set of keys and values of those keys which will trigger the plans. These keys are attributes of the *domain specific beliefs.* The GUI has facilities to specify such preconditions. Multiple preconditions can be specified which then are thought to be in disjunction. Postconditions are very similar and have a type associated with them for differentiating in achieved, unachievable and irrelevant cases. Returning parameters are the output keys passed to the instantiated team plan which have further information about the event that caused that plan to trigger. For example, FightFire plan is passed back information about location of FirePresent after the precondition is matched.

The second view is team organization hierarchy. The team hierarchy defines subteams as those teams that get more specialized down the tree. Thus the subteam FireEngines consists of engines that can fight chemical fire and engines that can fight electrical fires. The plan hierarchy nodes can be associated with a particular subteam via its name. As in this case only fireEngines subteam will be assigned to extinguish-fire subplan. When the domain expert wants to assign



Domain Belief	Attribute
FirePresent	Extinguished

Value: false

Key	Value
Extinguished	false

Returning Parameters: Location — Location

cond1

precon1

Fig. 2. Snapshot of tool showing team plan and user entering the preconditions and postconditions for ExtinguishFire subplan.

specific agents to specific tasks, this type of specification of teams and subteams can be useful. These can be used either as hints or constraints for role allocation.

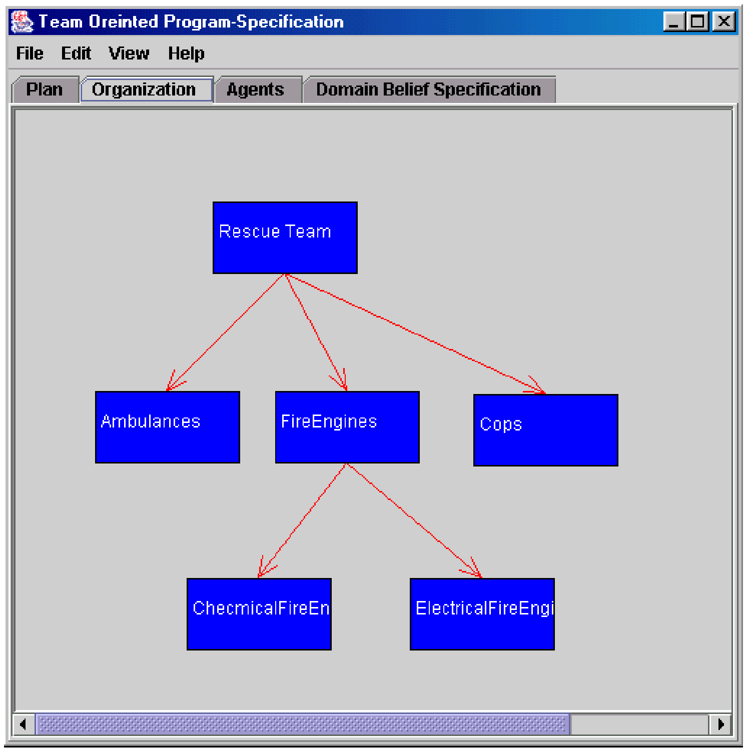


Fig. 3. Snapshot showing team organization hierarchy

The third view is a list of available domain agents and their capabilities. Capabilities are matched against requirements before agents are assigned to specific subteams. The last panel allows the user to specify the domain specific beliefs. We represent a domain specific belief as a key-value pair. For example, in the fire fighting domain, FirePresent belief has keys location and extinguished and corresponding values of where the fire is and its status, which the agent believes.

The last step is to save the TOPs specified graphically in XML documents as individual beliefs of agents. These beliefs consist of the team plans and the agents own capabilities. Hints or constraints for forming teams/subteams can be stored. Throughout the interface users are not allowed to input invalid specifications, such as specifying preconditions on non-existing keys. This simplifies validation procedure. Most of the XML generation is straight-forward. Currently the tool can handle the plan graphs which are tree-structured,, although it can be easily extended to any arbitrary plan graph.

5 Experiments

We have performed initial experiments in two domains. Each set of experiments and domain is aimed at testing one aspect of the Machinetta architecture.

5.1 Fire Fighting

In our first experiment, we used a simulator of a fire fighting domain[14]. The aim of this experiment was to include a human fire chief in the loop to help the team of intelligent agent fire fighters assign themselves to fires. The experiment was designed to show that the basic approach of identifying problems in team coordination and referring them to an expert was effective. The key parameter varied in this experiment is when the expert was brought into help. In particular, the *maxAsked* parameter controls the number of team members that should be offered a role before asking for expert input. If *maxAsked*= 0%, a proxy whose team member cannot (or will not) take on the role, the role will be immediately referred to an expert. If *maxAsked*= 100%, the algorithm ensures that all potentially capable team members are offered the role once before giving up. At the extreme, a special setting of *maxAsked*= ∞ means that proxies repeatedly pass the role amongst themselves (with each getting offered the role multiple times) without ever giving up. Varying *maxAsked* throughout this range produces distinct algorithms that produce different loads on role-allocation expert (i.e., the fire chief).

The fire chief interface consists of two frames. One frame shows a map of the city, displaying labeled markers for all of the fires that have been found, the positions of each fire brigade, and the location of the role each fire brigade is assigned to. The fire chief does not have direct access to the simulation state through the simulator itself, but is instead updated according to only the messages received by the fire chief's proxy. Therefore, the fire chief may be viewing a delayed picture of the simulation's progress. The other frame displays a list of all of the role-allocation tasks that have been allocated to the fire chief. By clicking on a task, the relevant capability information about each fire brigade is shown. The right-side window lists the fire brigades' distances to the fire, their water levels, and the roles they are currently performing. The fire chief can then view this data and find an appropriate agent to fulfill the role.

We conducted tests with three different fire chiefs. Each completed several practice runs with the simulation prior to experiments in order to minimize any learning effects. Each scenario was run for 100 time steps, with each step taking 30 seconds. The total data presented here represents 20 hours of run-time with a human in the loop.

Table 1 shows the team's domain-level performance across each experimental configuration. The scoring function measures how much of the city was destroyed by fire, with higher scores representing worse performance. The table shows the mean scores achieved, with the standard deviations in parentheses. Examining our two dimensions of interest, we can first compare the two rows to examine the effect of increasing the complexity of the coordination problem. In this case, increasing the number of fire brigades improves performance, as one might expect when adding resources while keeping the number of initial tasks fixed.

# Brigades	*maxAsked*= 0%	*maxAsked*= 100%	*maxAsked*= ∞
3	58(3.56)	73(16.97)	74(0.71)
10	52(19.09)	42(14.00)	73(4.24)

Table 1. Domain-level performance scores.

However, we can dig a little deeper and examine the effect of increasing complexity on the fire chief's performance. In the simpler configuration, asking the fire chief earlier (i.e., *maxAsked*= 0) improves performance, as the team gets a head start on exploiting the person's capabilities. On the other hand, in the more complex configuration, asking the fire chief earlier has the opposite effect. To better understand the effect of varying the point at which we assign roles to people, Table 2 presents some of the other statistics we gathered from these runs (mean values, with standard deviations in parentheses). With 3 brigades, if we count the mean number of roles taken on by the fire chief, we see that it stays roughly the same (401 vs. 407) across the two *maxAsked* settings. In this case, asking the fire chief sooner, allows the team to exploit the person's capabilities earlier, without much increase in his/her workload. On the other hand, with 10 brigades, the fire chief's mean role count increases from 563 to 716, so although the proxies ask the fire chief sooner, we are imposing a significant increase in the person's workload. Judging by the decreased average score in the bottom row of Table 1, the increased workload more than offsets the earlier exploitation of the person's capabilities. Thus, our experiments provide some evidence that increasing domain-level scale has significant consequences for the appropriate style of interaction with human team members.

Regardless of the variation of human behavior across scale, the data demonstrates that exploiting human capabilities can, in fact, improve overall team performance. We see this most clearly by examining the rightmost column of Table 1, which represents the results when the agents make all of the decisions. These scores are significantly worse than the leftmost data column, where the person is handed role-allocation roles immediately. Thus, the ability of our role-allocation algorithm to exploit the special coordination capabilities of people has provided a dramatic improvement in the performance of our team.

We can draw some additional conclusions about the heterogeneity introduced by people by clustering our statistics by person rather than by configuration. Each row in Table 3 represents the mean statistics of one of our three different fire chiefs. The "Tasks Performed" column counts the number of firefighting

# Brigs.	*max Asked*	Domain Roles	Fire Chief Roles	Tasks Performed	% Tasks Performed
3	0%	116 (7.12)	401 (51.81)	27 (6.55)	23.29 (6.51)
	100%	146 (33.94)	407 (54.45)	24 (6.36)	16.02 (0.63)
10	0%	103 (38.18)	864 (79.90)	67 (2.83)	14.49 (2.13)
	100%	98 (42.40)	563 (182.95)	41 (8.38)	48.06 (19.32)

Table 2. Role and fire-chief task metrics.

allocations performed by the fire chief, while the "% Performed" column measures that count against the number of total firefighting allocations assigned to the fire chief by the proxy architecture. Given the small sample size, we cannot draw any conclusions about a person's expected behavior. On the other hand, it *is* clear that we can expect a great deal of variance in behavior. For example, although fire chiefs A and B achieve roughly similar mean scores, they do so in very different ways. In fact, our proxies can expect fire chief A to be half as likely as fire chief B to respond to a task request. On the other hand, fire chief C is about equally likely as A to respond, and C performs roughly the same number of tasks as B, yet C achieves only half the score as the other two. Thus, it appears unlikely that we can easily classify people's capabilities, since, for even the relatively few dimensions measured here, our human fire chiefs show no generalizable characteristics.

Fire Chief	Score	Tasks Performed	% Performed
A	0.61	25	28%
B	0.59	40	56%
C	0.31	42	32%

Table 3. Statistics for each Fire Chief.

5.2 Recharging 100 Robots

In our second domain, we have a large number of sensor robots (CSRs) distributed in some environment over and extended period. Over time the batteries on the CSR robots run down and they need to be collected for recharging by a CHR robot. The CSR robot is led to the recharging station by the CHR robot, hence must have some remaining battery power to be recharged. This scenario is part of DARPA's Software for Distributed Robotics program. Proxies running on CHR robots must cooperatively work out which CHR collects which CSR. In this set of experiments we aim to better establish the properties of the autonomous role allocation algorithm.

We have conducted several experiments in order to evaluate our approach, using a simulation of the distributed robotics domain. The simulator represents the building as a grid and the CHRs are able to move from grid location to grid location, pick up CSRs and recharge CSRs by moving them to a recharge station. While the details of robot control are not simulated, the uncertainty CHRs have about their position is modelled using a localization algorithm very similar to those used on real CHRs. In particular, the localization algorithm uses a well known markovian localization method [4], based on simulated landmarks in the building. Battery level in the CSRs decrease with uncertainty, thus it is not possible to predict its dynamic during the task execution.

We used two different kinds of simulation set up. In the first one, the experiments are conducted without using the proxies for the role assignment. The role allocation approach is implemented in a software module inside the simulator. In the second setting the proxies have been connected to the simulator and execute the same approach for the role assignment. While in the first set of experiments we mainly focused on investigating how different parameter settings for the environment affect the performance of our approach, the second experimental setting is used to validate the obtained results using the proxies framework.

In the first set of experiments, we tested four different algorithms: the allocation algorithm described in [14], an extension of this algorithm to handle dynamic capability estimation, a mechanism for the uncertainty handling, and finally the combination of these two extensions. We decided to vary the amount of CHRs that can have a degradation on their localization capability during the experiments and investigate how this parameter affects the different algorithms performance.

For the first set of experiments we used an environment with 24 CHRs and 47 CSRs and each experiment is 6000 simulation steps long. For each different parameter setting we performed five repetitions. The results obtained are reported in table 4 and in table 5. Table 4 show the results when five CHRs experience problems in their localization capability while table 5 reports the result with ten. In each table the first column shows the algorithm used, the second column shows the average battery level of CSRs over time. The third column shows the average of the minimum battery level of all the CSRs over time. In both the second and third columns, the averages exclude the battery levels of robots that have failed. The fourth column of the tables, show the number of CSR that completely failed, i.e., the number of CSRs whose battery level falls to 0. Finally the last column shows the standard deviation computed over the five repetitions.

The results show that the overall performance of the team is negatively affected, when more CHRs have their localization capability degraded. When comparing results obtained using the overload handling algorithm with the basic algorithm, the number of failed CSRs were lower while both the average battery level and the average of the minimum battery level were improved. The improvement is similar both for the case when five and ten CHRs can have a degrading localization capability. Moreover, the overload handling algorithm re-

Algorithm	Avg B. L.	Min B. L.	Fail	σ
Basic	0.644672	0.195902	13.6	2.8
Ovl Handl.	0.65997	0.223343	11.8	0.97
Unc Handl.	0.693892	0.229101	12.2	0.75
Ovl and Unc	0.684224	0.241805	9.8	1.47

Table 4. Results for five CHR with localization problems

Algorithm	Avg B. L.	Min B. L.	Fail	σ
Basic	0.633321	0.17654	20.6	2.58
Ovl Handl.	0.65293	0.215206	17.6	1.85
Unc Handl.	0.691214	0.221129	15.8	2.64
Ovl and Unc	0.691896	0.219198	16.2	2.79

Table 5. Results for ten CHR with localization problems

Algorithm	Avg B. L.	Min B. L.	Fail	σ
Unc Hnd. 5	0.695983	0.238347	14.6	1.36
Unc Hnd. 10	0.699272	0.237091	17.6	2.87

Table 6. Results for the limited exchange

Algorithm	Avg B. L.	Min B. L.	Fail	σ
Unc Handl.	0.699902	0.236674	12.5	1.65

Table 7. Results for the distributed setting

sults in a lower standard deviation from the average failure value, showing a better adaption to problematic situations.

Also the algorithm for uncertainty handling seems to improve the performance for the overall team. In particular for the result reported in table 4 we have a very low standard deviation, similar to the overload handling mechanism. However when the number of CHR that can have localization problems is higher we have actually a higher standard deviation but still acceptable results. The results reported in table 4 and 5 for the uncertainty handling algorithm, are obtained assuming that each CHR can ask and have a response at each simulation step from all its team mates when trying to exchange a role. This is a very strong assumption and it is not likely to be met in the real application. Therefore we performed an experiment limiting the number of team mates that can be queried during each time step. In table 6 we report the results for this set of experiments. The first row of the table refers to the case where five CHR can have their localization capability degraded, while the second row reports the results for ten. In both cases, the results are worst if compared with the respective row of table 4 and 5. These experiments show that the discussed approach could not be effective enough for our reference scenario, where the assumption made in the previous experiments could easily not be met.

In the second experimental setting we connected the proxies to the simulator. We decided to test the algorithm for the uncertainty handling, when five CHRs can have a degradation in their localization capability. All the parameters described in the previous set of experiments are used also in this set, except for the number of repetition that in this case is not five but two. These experiments have been conducted in order to see how the overall performance of the algorithm could be affected using the actual proxy framework. In particular for our scenario, a very important issue is the conflict that can possibly arise among the proxies' information on the actual world state, due to the asinchronicity of the message passing approach. The results reported in table 7 show that the algorithm performance seems not to be heavily affected by this issue, however the small number of experiments conducted does not allow to draw a statistically significant conclusion, and further investigations need to be done.

6 Related Work

Proxy-based integration architectures are not a new concept, however no previous architecture has been explicitly designed to have robots, agents and people in the same team. Jennings's GRATE* [5] uses a teamwork module, implementing a model of cooperation based on the joint intentions framework. Each agent has its own *cooperation level* module that negotiates involvement in a joint task and maintains information about its own and other agents' involvement in joint goals. Jones [6], Fong [16], Kortenkamp[8] and others have worked on improving collaboration between groups of robots and a single person, though these approaches to robotics teams have not explicitly used proxies. The Electric Elves project was the first human-agent collaboration architecture to include both proxies and adjustable autonomy[2]. COLLAGEN [13] uses a proxy architecture for collaboration between a single agent and user. Payne et al[19] illustrate how variance in an agent's interaction style with humans affects performance in domain tasks. Tidhar [20] used the term "team-oriented programming" to describe a conceptual framework for specifying team behaviors based on mutual beliefs and joint plans, coupled with organizational structures. His framework also addressed the issue of team selection [20] — team selection matches the "skills" required for executing a team plan against agents that have those skills.

7 Conclusions and Future Work

As seen in both domains, Machinetta shows promise in allowing complex teams to tackle the challenge of effective coordination. The main advantages to our approach become apparent when dealing with teams that display one or a combination of the characteristics: large scale, dynamic environment, and integration of humans. By connecting the Machinetta proxies with the graphical development tool for constructing team plans, the TOP programmer gains a good idea of what is going on in the plan and how to make effective changes in it in order to have the team behave more desirably. In the future, we plan on extending the features of both the graphical planning tool and Machinetta itself, while keeping the framework generalizable.

References

1. H. Chalupsky, Y. Gil, C. Knoblock, K. Lerman, J. Oh, D. Pynadath, T. Russ, and M. Tambe. Electric Elves: Applying agent technology to support human organizations. In *International Conference on Innovative Applications of AI*, pages 51–58, 2001.
2. Hans Chalupsky, Yolanda Gil, Craig A. Knoblock, Kristina Lerman, Jean Oh, David V. Pynadath, Thomas A. Russ, and Milind Tambe. Electric Elves: Agent technology for supporting human organizations. *AI Magazine*, 23(2):11–24, 2002.
3. Philip R. Cohen and Hector J. Levesque. Teamwork. *Nous*, 25(4):487–512, 1991.
4. Dieter Fox, Wolfram Burgard, and Sebastian Thrun. Markov localization for mobile robots in dynamic environments. *Journal of Artificial Intelligence Research*, 11:391–427, 1999.
5. N. Jennings. The archon systems and its applications. Project Report, 1995.
6. Henry L. Jones, Stephen M. Rock, Dennis Burns, and Steve Morris. Autonomous robots in swat applications: Research, design, and operations challenges. In *AUVSI '02*, 2002.
7. S. Kelso. *Dynamic Patterns: the self-organization of brain and behavior.* The MIT Press, 1995.
8. D. Kortenkamp, D. Schreckenghost, and C. Martin. User interaction with multi-robot systems. In *Proceedings of Workshop on Multi-Robot Systems*, 2002.
9. Dave Mulsiner and Barney Pell. Call for papers: AAAI spring symposium on adjustable autonomy. www.aaai.org, 1999.
10. David Pynadath and Milind Tambe. Multiagent teamwork: Analyzing the optimality and complexity of key theories and models. In *First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'02)*, 2002.

11. David V. Pynadath and Milind Tambe. An automated teamwork infrastructure for heterogeneous software agents and humans. *Journal of Autonomous Agents and Multi-Agent Systems, Special Issue on Infrastructure and Requirements for Building Research Grade Multi-Agent Systems*, page to appear, 2002.
12. D.V. Pynadath, M. Tambe, N. Chauvat, and L. Cavedon. Toward team-oriented programming. In *Intelligent Agents VI: Agent Theories, Architectures, and Languages*, pages 233–247, 1999.
13. C. Rich and C. Sidner. COLLAGEN: When agents collaborate with people. In *Proceedings of the International Conference on Autonomous Agents (Agents'97)*, 1997.
14. P. Scerri, D. V. Pynadath, L. Johnson, Rosenbloom P., N. Schurr, M Si, and M. Tambe. A prototype infrastructure for distributed robot-agent-person teams. In *The Second International Joint Conference on Autonomous Agents and Multiagent Systems*, 2003.
15. D. Schreckenghost, C. Thronesbery, P. Bonasso, D. Kortenkamp, and C. Martin. Intelligent control of life support for space missions. *IEEE Intelligent Systems Magazine*, 2002.
16. C. Thorpe T. Fong and C. Baur. Advanced interfaces for vehicle teleoperation: collaborative control, sensor fusion displays, and web-based tools. In *Vehicle Teleoperation Interfaces Workshop, IEEE International Conference on Robotics and Automation*, San Fransisco, CA, April 2000.
17. Milind Tambe. Agent architectures for flexible, practical teamwork. *National Conference on AI (AAAI97)*, pages 22–28, 1997.
18. Milind Tambe, Wei-Min Shen, Maja Mataric, David Pynadath, Dani Goldberg, Pragnesh Jay Modi, Zhun Qiu, and Behnam Salemi. Teamwork in cyberspace: using TEAMCORE to make agents team-ready. In *AAAI Spring Symposium on agents in cyberspace*, 1999.
19. Katia Sycara Terry Payne and Michael Lewis. Varying the user interaction within multiagent systems. In *Agents'00*, pages 412–418, 2000.
20. G. Tidhar, A.S. Rao, and E.A. Sonenberg. Guided team selection. In *Proceedings of the Second International Conference on Multi-Agent Systems*, 1996.