# Towards Collaborative Task and Team Maintenance

Gal A. Kaminka Ari Yakir Dan Erusalimchik Nirom Cohen-Nov*
The MAVERICK Group
Computer Science Department
Bar Ilan University, Israel
{galk,yakira,yeda,cohennn}@cs.biu.ac.il

## ABSTRACT

There is significant interest in modeling teamwork in agents. In recent years, it has become widely accepted that it is possible to separate teamwork from taskwork, providing support for domain-independent teamwork at an architectural level, using teamwork models. However, existing teamwork models (both in theory and practice) focus almost exclusively on achievement goals, and ignore *maintenance goals*, where the value of a proposition is to be maintained over time. Such maintenance goals exist both in taskwork (i.e., agents take actions to maintain a condition while a task is executing), as well as in teamwork (i.e., agents take actions to maintain the team). This paper presents mechanisms for collaborative maintenance in both taskwork and teamwork, allowing for flexible selection of the maintenance protocol. The mechanism is integrated and evaluated in two teamwork architectures for situated agent teams: DIESEL , an implemented teamwork and taskwork architecture, built on top of Soar, and BITE , an architecture for physical behavior-based robots. We provide details of these implementations, and the results from experiments demonstrating the benefits of support for collaborative maintenance processes, in several dynamic rich domains. We show that the use of collaborative maintenance leads to significant improvement in task performance in all domains.

## 1. INTRODUCTION

In recent years, it has become widely accepted that it is possible to use machine-executable teamwork models to automate collaboration at an architectural level. Such models separate teamwork from taskwork, allowing the deployer of a team of agents to focus her efforts on programming the skills and knowledge necessary for the specific task. Executable teamwork models have been utilized successfully in synthetic agents for training and simulation [15], robotics [14, 7], industrial distributed systems [6], and collaborative user interface [12].

However, existing models only account for a subset of phenomena associated with teamwork. Specifically, existing teamwork models focus almost exclusive on *achievement goals*, where the value of a proposition is to be changed from its current settings to another. Agents form a team and agree on a task to be executed (goal to be reached, i.e., proposition to hold in some future state), and then dissolve the team once the task is completed. Sequences of tasks are carried out by constant dissolving and re-formation of the team in question, per task [16].

Human and synthetic teams, however, must also tackle *maintenance goals*, where the value of a proposition is to be maintained over time. Such maintenance goals exist both in taskwork (i.e., agents take collaborative actions to maintain a condition while a task is executing), as well as in teamwork (i.e., agents take actions to maintain the team). Examples of maintenance goals in teamwork include robust service maintenance [10, 9] and continual task allocation [14]. Examples of maintenance goals in taskwork includes continual information sharing and monitoring for robotic formations [1]. Architectures that only address achievement goals are not sufficient for handling maintenance goals.

We use an example of taskwork maintenance to illustrate. Here, a team of agents consists of multiple agents that follows a leader at a distance. This is a simplified version of familiar robotic formation-maintenance tasks (e.g., [1]), or the convoy task (e.g., [2]). Existing teamwork architectures, based on teamwork theory [2]), would have the followers communicate with the leader (or otherwise monitor it) to establish mutual belief that the distance is correct or incorrect (goal achieved or unachieved). Based on failures, corrective actions could be taken, which in essence *react* to the failures of the robots. Similar cases occur in maintenance of teamwork.

But a different—and more efficient—approach would have the followers and leader take *proactive* actions to maintain the distance, before it becomes too great. For example, the leader may communicate its position to its followers, to help them speed-up or slow-down incrementally, such that the distance never goes out of bounds. The point is that here communications occur while maintaining a condition, rather than when it unmaintained. With very few exceptions (see Section 2), existing teamwork theory and teamwork architectures do not account for such communications.

This paper addresses maintenance goals in situated agent teams, from an architectural perspective. First, we show how to integrate maintenance conditions into a behavior-hierarchy, used for controlling each individual agents. Building on this infrastructure, we present several contributions: (i) a mechanism for *collaborative* maintenance of taskwork conditions by team-members, allowing them to flexibly select different maintenance protocols; (ii) the reuse of this mechanism to maintain teamwork-structure conditions; and (iii) the integration of this mechanism in two teamwork architectures, for different tasks.

We evaluate these contributions in two different teamwork architectures, and in different environments: DIESEL , implemented on

*This research was supported by ISF Grant #1211/04.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*AAMAS'07* May 14–18 2007, Honolulu, Hawai'i, USA.
Copyright 2007 IFAAMAS .



978-81-904262-7-5 (RPS) ©2007 IFAAMAS

top of Soar [13] and used in virtual worlds, and BITE [7], used for controlling teams of physical robots. We report on experiments evaluating the use of collaboratively-maintained maintenance conditions in contrast to existing approaches, using either achievement goals, or individual maintenance processes. We show that the collaborative maintenance mechanism leads to significantly improved performance in different tasks and domains.

## 2. RELATED WORK

Most teamwork architectures to date have only allowed for achievement goals, and we therefore focus here only on those that have addressed maintenance goals to some extent.

Kumar and Cohen [10, 9] extended the theory of Joint Intentions to include maintenance. They define the goal of maintaining $p$ as follows: *if the agent does not believe* $p$*, it will adopt the goal that* $p$ *be eventually true*. The maintenance goal is persistent (PMtG) if $p$ is believed false at least until the agent either believes that it is impossible to maintain $p$ or that the maintenance goal is irrelevant.

While we build on the theoretical developments of [10, 9], our work differs significantly. First, we extend maintenance of team structure to hierarchical teams, including team-subteam relations. We also address goal maintenance in hierarchical task decomposition. Second, our implementations in DIESEL and BITE allow for arbitrary, context-dependent protocols (some by using communications, some not) for collaborative goal maintenance, where Kumar et al. have used a fixed protocol. Finally, while Kumar and Cohen's work has been applied to teams of web services, our focus is on modeling synthetic humans in virtual environments, and in robotic tasks.

STEAM [15], implemented in Soar [13], focuses for the most part on achievement goals. However, a first step towards extending STEAM towards maintenance goals was introduced in [16]. Here, maintenance is addressed through persistence in the commitment of agents to the team, while executing a task. Four categories of teams are introduced: PTPM, a persistent team consisting of persistent members; PTNM, a persistent team consisting of non-persistent members; NTPM, a temporary form of a team consisting of persistent members; and NTNM, a temporary form of a team consisting of non-persistent members. This work was the first to discuss reorganization (team hierarchy maintenance) in a team, i.e., PTNM.

To enable persistent teams in STEAM, agents individually reason about expected team utilities of future team states, to decide on how to best maintain the team in face of intermittent failures in teamwork. DIESEL , described in this paper, deals with PTPM teams, i.e., persistence of team structure. We refer to this as teamwork maintenance. However, in contrast to [16], DIESEL and BITE also address collaborative maintenance in tasks (*taskwork maintenance*). Moreover, we propose a single mechanisms for both, and offer flexibility to the designer and agents in deciding on protocols and behaviors to be used proactively and reactively.

CAST [17] addressed the issue of proactive information exchange among teammates, using an algorithm called DIARG, based on petri net structures. CAST shows the importance of team communication regarding information that might assist task achievement for individual members in a proactive manner, and aim to reduce communication. This approach, based on the theory of Joint Intentions, does not include maintenance of goals. In particular, CAST's communications focus on informing other teammates of discovered facts that may trigger preconditions. The use of communications (or other actions) to maintain currently existing tasks is not addressed.

ALLIANCE [14] is a behavior-based control architecture focused on robustness, in which robots dynamically allocate and re-allocate themselves to tasks, based on their failures and those of their teammates. ALLIANCE offers continual dynamic task allocation facilities, which allocate and re-allocate tasks to agents while they are collaborating. It uses fixed teams, in the sense that addition and removal of robots from the team is handled by human intervention and it assumes that robots can monitor their own actions, and those of others. Our work differs in that we focus on maintenance not only of assignment of agents to tasks, but also of the joint execution itself.

## 3. MAINTENANCE IN TEAMWORK

We propose a new architectural mechanism that allows the automation of maintenance both of the team structure and of the behavioral structure. Our architecture extends structures common to situated agent architectures. It is composed of four structures:

1. A *behavior graph* (*recipe*), that defines the decomposition and temporal constraints on the task-oriented behavior of the agents.
2. A *team hierarchy* that defines the organizational structure and chain of command.
3. A set of domain-independent reusable *task-maintenance* behaviors, referenced by the recipe.
4. A set of domain-independent reusable *team-maintenance* protocols, referenced by the team hierarchy.

Structure 1 is common to many situated agent architectures (e.g., [11, 5, 13]). Structure 2 is an addition, which appears only in architectures managing teamwork, such as STEAM [15], and previous versions of BITE [7]. These two structures are described in Section 3.1. The last two structures (3 and 4) are unique to our approach, and described in Section 3.2.

### 3.1 Common Structures in Situated Agent Teams

We begin by a brief overview of situated agent control, upon which the maintenance mechanism is based. Each agent is controlled by a connected, directed graph, that defines a structure by which agents achieve their goals. Each node in the graph is a controller, called a *behavior* (in Soar, *operator*). Each behavior has *preconditions* which enable its selection (the agent can select between enabled behaviors), *termination conditions* (which determine when its execution must be stopped, if previously selected) and application code containing the actual code for execution while the behavior is running.

Edges in the graph have two types: Decomposition edges specify how a controller can be decomposed into sub-controllers (executing sub-tasks); sequential edges specify the temporal ordering of behavior execution. The graph is arranged such that cycles are allowed along sequential links, but not along decomposition links: A node can follow itself temporally, but cannot be its own ancestor. We allow for reactivity: A behavior is not always selected when its predecessor terminates. Instead, the agent's control process may choose to select a different behavior that is selectable (as long as it is a first child of an active parent). The graph is referred to as a *recipe* [4], a *plan hierarchy* [15], or *behavior graph* [7]. We will use these terms interchangeably.

An example recipe is shown in Figure 1. Vertical edges signify decomposition (i.e., from a behavior to sub-behaviors needed to execute it); horizontal edges signify temporal ordering, from a

behavior to those that should ideally immediately follow it. Here, the recipe has two nodes called explore-decision and explore-movement. As a rule, we read recipes left to right: Thus explore-decision is considered the first child. Only once it terminates, can explore-movement be selected. explore-decision has two first children, i.e., two alternative decompositions. Only one of them is to be selected for execution.

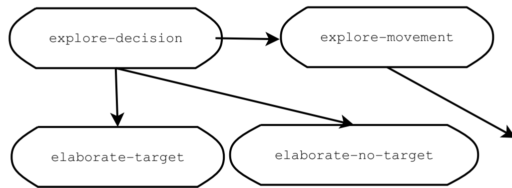



**Figure 1: An example recipe.**

To maintain knowledge of the organizational structure of the team, a second structure—*team hierarchy*—is used. This structure is a tree, in which internal nodes represent subteams, and leaves denote individual agents. Edges represent team-subteam relations. Although teamwork architectures differ in how they achieve this, they utilize this structure to automatically determine which agents are parts of which subteam, so that when a behavior is selected by an agent, this selection is automatically coordinated with the other members of the team. This is done by maintaining a pointer from each behavior, to the team-hierarchy node that is associated with it. Mechanisms for such automated coordination are described elsewhere ([6, 15, 7]) and will not be discussed here further. Both DIESEL and BITE , described here, take a similar approach.

## 3.2 Collaborative Maintenance Behaviors

To facilitate maintenance, we add a third type of condition to the preconditions and termination conditions already associated with each behavior. *Maintenance conditions* are propositions whose value is to be maintained throughout the lifetime of the behavior. Such conditions can be a conjunction or disjunction of predicates (referred to as events).

Maintenance conditions can typically be maintained in one or two ways: By taking proactive actions to maintain the condition true; and by taking reactive actions when the condition becomes false. The latter option (reactive maintenance) is similar in spirit to the use of a sequence of achievement actions in order to maintain a condition. However, the former type has no such translation. Thus the two types are different, and indeed, must be distinguished in the definition of the behavior.

To maintain the condition, we allow the definition of maintenance behaviors, which are to be associated with specific maintenance conditions, and with specific types of maintenance (reactive or proactive). If a reactive maintenance behavior is defined, then the architecture will trigger it once the maintenance condition breaks. If a proactive maintenance behavior is defined, the architecture will trigger it once the behavior is selected, so that it execute while the original behavior is running.

Though the use of maintenance conditions in integrated architectures is rare, the key novelty described in this paper is the ability to tie specific *team behaviors* to these conditions. The behaviors will be triggered automatically by DIESEL or BITE , to be executed jointly, in a coordinated manner, by the team or subteam associated with the behavior. It thus becomes possible to collaboratively maintain a condition, rather than individually.

For example, suppose a behavior $B$ that moves the agents around has a maintenance condition on it to maintain visual tracking of the leader. Because $B$ is a team behavior, it will be executed by the leader and the follower jointly. As a result, both leader and follower are mutually responsible for maintaining the condition. The maintenance behavior $M$ then becomes itself a team-behavior, to be executed jointly by the leader and follower even as they are executing $B$ (i.e., move around). An example of such a maintenance behavior may have the leader continually communicate its current position, and the follower orienting itself towards this position.

**Teamwork maintenance.** Just as task-execution behaviors can have associated maintenance conditions, so can the team hierarchy be maintained by the use of team-maintenance conditions. As in the behavior hierarchy, these conditions are a set of conjunctions and disjunctions of predicates (referred to as events), needed to be maintained or denied throughout the execution of a task. Since maintenance operators act in order to maintain a possible team state, they are suited to allow team reconfiguration, all under the same teamwork mechanism. For example, if during the execution of a recipe sub-tree it is critical to maintain the number of teammates in the group fixed, such a team-maintenance condition could be easily defined, and the teamwork mechanism, can act in turn if such a condition fails, by joining a new team, recruiting new agents or even merging two teams. All whilst continuing execution of the mission.

# 4. TWO IMPLEMENTATIONS

We have implemented the maintenance mechanism described above in two different architectures. DIESEL , built on top of the Soar integrated cognitive architecture [13], was built from the ground up with collaborative maintenance in mind. The other implementation revisited the BITE behavior-based multi-robot architecture [7], and extended it to support maintenance behaviors. The two architectures were used in very different settings, and it is thus evidence of the generality of the mechanism that it was successfully integrated in both. We describe the implementations below.

## 4.1 DIESEL

We use Soar for the implementation of our architecture. While a comprehensive description of Soar is beyond the scope of this paper, we provide a brief overview here. Soar is a general cognitive architecture for developing systems that exhibit intelligent behavior [13]. It is a rule-based (production rules) language. Soar uses parallel, associative memory (the rules), along side a graph-structured global working memory, which all rules can access and modify. All knowledge relevant to the current problem is identified and brought to bear via the rules. These trigger the proposal, selection, execution, and termination of operators, which are also implemented using rules. Soar has belief maintenance through a computationally inexpensive truth maintenance algorithm. Deliberation in Soar is mediated by preferences, which allow agents to bring knowledge to bear in order to make decisions. Soar recognizes conflicts (impasses) in selection knowledge and automatically creates subgoals (new state spaces) to resolve the impasses. Once an agent comes to a decision that resolves an impasse, it summarizes and generalizes the reasoning during the impasse. This summary information can be used by an integrated explanation-based learning mechanism, to automatically create new rules that chunk problem-solving results for future use.

The DIESEL teamwork architecture is implemented as a set of Soar rules, running as a separate mechanism on top of the underlying Soar architecture. The complete DIESEL engine is composed of approximately 100 Soar productions (IF-THEN rules), of which 23 implement the recipe structure and associated mechanisms, and 25 implement the basic teamwork capabilities (including team hierar-

chy, collaborative maintenance mechanisms, and basic communications).

In DIESEL , maintenance behaviors (whether collaborative or individual) are triggered based on events (termination conditions), with no regard to the behavior (in Soar, operator) currently executing. Thus once the programmer writes a maintenance operator for a predicate $p$, the operator will be triggered to maintain $p$ regardless of which operator has $p$ as a maintenance condition. This design choice has the advantage that operators for maintaining predicates of interest can be easily re-used within the agent code. However, the disadvantage is that this leaves no room for flexible selection of maintenance operators: The same maintenance operators would be proposed, regardless of the execution context (current running operators). BITE takes the inverse approach, as described below.

DIESEL has been applied in two separate virtual environments, and for different tasks. It has been used in Soar agents for the GameBots environment [8], an adversarial game environment that enables qualitative comparison of different control techniques. Figure 2 shows a screen-shot of Soar agents in the GameBots domain, running DIESEL .

**Figure 2: Soar agents in the GameBots environment, running DIESEL . Each agent has limited field of view and range, and may move about, turn, grab objects, etc.**

## 4.2 BITE

BITE is a behavior-based teamwork architectures for robots. Previous versions of it (without the maintenance mechanisms) were used in controlling Sony AIBO robots moving in formation [7] (Figure 3). The version we use is implemented for the player-stage system, a de-facto standard API for controlling different types of physical and simulated robots [3], rather than only AIBOs; code running in the simulation can be used with few modifications on the physical robots. Figure 4 shows a view of BITE -controlled robots moving in formation, in the player-stage simulator.

The key novelty in BITE is its micro-kernel design, in which all protocols for coordinating multiple robots are taken out of the system, and are made into a library from which the user (the deployer of a robot team) can choose protocols, mixing them within the same task (but not within the same behavior) as she sees fit. Thus in BITE , unlike previous architectures, the designer can tell a team of robots to use a bidding protocol to decide on their assignments to roles in a formation, and a different protocol to assign themselves other tasks. These protocols and coordination procedures are grouped together as *social behaviors*.

The use of BITE in formation-maintenance tasks has brought up the need for the integration of a maintenance mechanism within the architecture. Most formation-maintenance algorithms rely on visual tracking of a leading robot, by its followers, to maintain a fixed

**Figure 3: Sony AIBO robots moving in formation.**

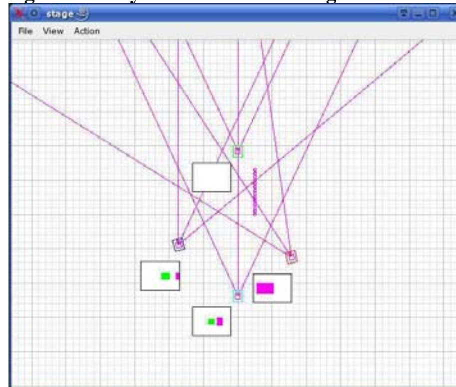


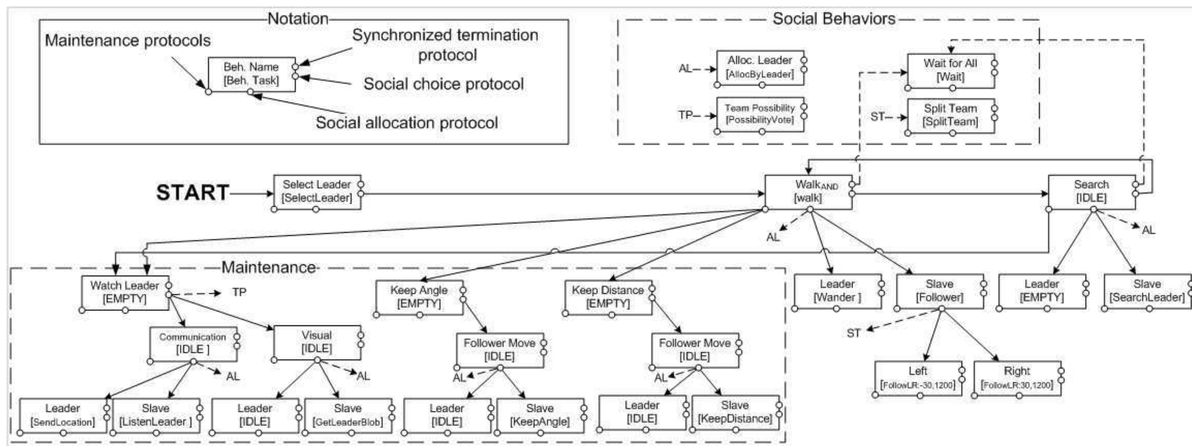

**Figure 4: Robots running BITE in the player-stage API. The robots form a diamond. The lines mark visual field of view. Boxes with filled blocks show the colors perceived by each robot.**

distance and angle to the leader (see, for instance [1]). In earlier versions of BITE , which only allowed for collaborative achievement goals, such maintenance was always done ad-hoc, within the controlling behaviors. As a result, formation-maintenance in BITE did not exploit the automated teamwork mechanisms in the architecture: The leading robot took no part in maintaining its distance from its followers.

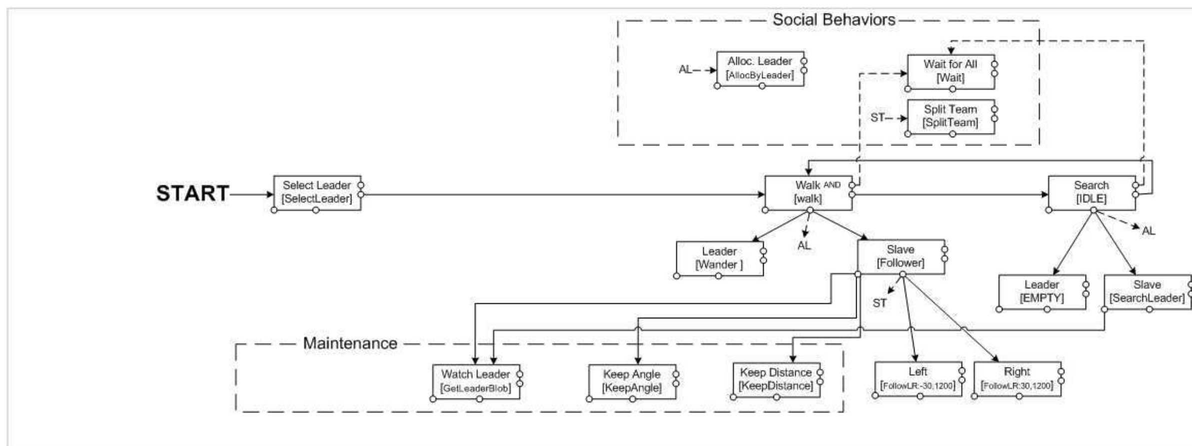
However, once the maintenance mechanism is introduced into BITE , then all of a sudden a range of novel possibilities emerge. For instance, it is now possible to write the formation maintenance task in terms of reusable angle, and distance maintenance behaviors. Moreover, it is now possible to have BITE automatically have the leader communicate its position or take other actions, such that its followers can track it more easily. Moreover, the implementation in BITE allows to easily see the difference between individual and collaborative maintenance.

Figure 5 shows the behavior graph, social behaviors, and maintenance behaviors in BITE , for the formation-maintenance task. Figure 5(a) shows the behavior structures when using the collaborative maintenance mechanism. Figure 5(b) shows the same task, but with individual maintenance. The upper-left corner of Figure 5(a) explains the notation: Solid lines indicate structural links (decomposition and sequence constraints); dashed lines indicate pointers to social behaviors, to be triggered whenever BITE triggers automated coordination (see [7] for details, which are outside the scope of this paper).

The behavior-graph for the task is in fact quite small. There are three top-level *team* behaviors: Select-leader, Walk and Search.




(a) BITE **with collaborative maintenance behaviors.**




(b) BITE **with individual maintenance behaviors.**

**Figure 5: Collaborative and individual maintenance behaviors in** BITE **.**

Select-leader is the first behavior, where a leader for the formation is selected by application-specific code. Then, the team of robots jointly selects Walk to begin the movement. The joint selection of Walk is carried out automatically by an appropriate social behavior (separated by a dashed box, in the figures). If one of the robots fails to monitor the leader, then the team will jointly terminate this behavior and select Search to re-organize. As possible decompositions of each of these team behaviors, the robots may individually select behaviors based on their role.

In Figure 5(b), maintenance of the leader in sight, and maintenance of distance and angle to the leader, are all done through individual maintenance behaviors (sf Watch Leader, Keep Angle, Keep Distance in the figure). The fact that these are individual maintenance behaviors is established structurally: They are tied to the Slave behavior, which is executed individually, as a decomposition of Walk. Here, it is strictly up to the followers, executing these behaviors, to maintain the conditions of the formation.

In Figure 5(a) these maintenance behaviors are tied not to the individual follower behaviors, but to the team behavior Walk. Automatically, the maintenance behaviors are treated as team behaviors, and their execution is thus synchronized (by the same mechanisms that synchronize execution of the regular behaviors). Their elevation to the status of collaborative maintenance behaviors offers new possibilities for maintenance, e.g., the use of communication by the leader. These increased options are the reason for the many additional behaviors in Figure 5(a) as compared to 5(b).

## 5. EVALUATION

To evaluate the contribution offered by the introduction of collaborative maintenance in DIESEL and BITE , we conducted a number of experiments with both architectures, each in its respective application environment. Together, the application of the mechanisms to a variety of domains provide evidence for its usefulness as a general architectural component.

The experiments were designed to answer a number of hypothe-

ses. A first set of experiments (Section 5.1) provides evidence that a proactive maintenance mechanism (which acts to preserve the value of the maintenance condition before it is falsified) is preferable to the use of reactive responses to the breaking of the maintenance condition. A second set of experiments (Section 5.2) then considers the hypotheses that collaborative maintenance is preferable, and leads to improved results, compared to individual maintenance (even using the same mechanism). A final set of experiments (Section 5.3) demonstrates how team reconfiguration occurs using the maintenance mechanism with the team hierarchy, rather than the behavior graph.

## 5.1 Individual Achievement vs. Maintenance

The first set of experiments focused on establishing the importance of a proactive maintenance mechanism (even for individual execution), compared to the reactive use of a sequence of achievement actions to correct the value of maintenance condition. These experiments were carried out with DIESEL .

We built a small two-agent team in the GameBots domain [8]. In all experiments, we used the same recipe (Figure 1), with minor changes needed for each scenario discussed. The recipe consists of exploration and movement. During the exploration phase (behavior *explore-decision*), one of the two child behaviors can be proposed: *elaborate-no-target* in case there is no available target present, and *elaborate-target* in case there are one or more. In the first case, the agent will tilt its pan-zoom camera, scan or rotate, and in the second case, a behavior will summarize target data, and propose all available options. *explore-decision*'s termination condition is that a target has been selected. Respectively, this is *explore-movement*'s precondition. In this case, a child behavior will be in charge of all movement actions taken by the agent in order to reallocate itself to a given target location.

Our first experiment examines DIESEL 's explicit support for maintenance goals, using the idea of task-maintenance behaviors that execute in parallel to the task-achievement behaviors. In this experiment, two agents are placed side by side on one end of a long corridor, closed off by walls at both ends. One agent is a leader, the other a follower. The leader runs until reaching the wall and then runs back. The follower's task is to run after the leader. In the individual achievement case, the follower will look for the leader whenever it loses sight of it. In the individual maintenance case, the follower will continuously orient itself such that the leader is centered in its field of view (regardless of the direction in which the follower is running).

We are prohibiting any communications at this stage, since the task is purely individual for now. The follower will scan until it sees the leader, and run towards it. During each time tick, if the follower agent sees the leading agent, an internal event (*see-leader*) is fired and logged. Each configuration was run 10 times. In each run, the simulation's duration was about two minutes in real-time, approximately 9000 decision-sense-act Soar cycles and fifty seconds in Unreal Tournament clock-time.

The results from this experiment are summarized in Table 1. We use two measures: The first column measures the percentage of time (averaged across the ten trials) in which the leader was seen, i.e., the maintenance conditions was indeed maintained. The second column measures the number of behavior switches taking place on average, in each run. A reduced number of behavior switches is one desired outcome of using maintenance behaviors in parallel to task execution. This reduces thrashing and allows for greater use of context in Soar and similar architectures. The three rows correspond to the results for the achievement configuration, the proactive maintenance configuration, and a one-tailed t-test of the statistical

| | % Time leader in sight | # Behavior switches |
|---|---|---|
| Achievement | 50% | 6 |
| Maintenance | **66%** | **4.1** |
| one-tailed t-test | 0.01 | 0.001 |

**Table 1: Individual achievement (reactive maintenance) compared to individual proactive maintenance.**

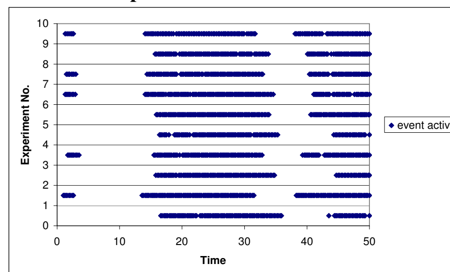



**Figure 6: *see-leader* event logged by the follower agent. No maintenance conditions.**

significance of the results.

The results clearly demonstrate the need for a proactive maintenance mechanism. In both measures, the agents using maintenance (even individually) have come up significantly on top, compared to the achievement-only configuration.

Figure 6 shows the same results, from a different perspective. It shows the event's occurrence during the ten simulation runs. In the figure, The X-axis shows the time. The Y-axis separates the ten trials: Each dot shows the presence of the see-leader event in memory, at the give time, for the given trial. The figure shows that in all the experiments conducted without maintenance, after a short period of time, the follower lost the leader. This is due to the change in direction of the leading agent (back to the start location after reaching the wall) which occurred during the follower's movement. In addition, sometimes when the follower agent locates the leader right away, it is only for a short period of time. This is due to the fact that the leader agent chose its target and began moving towards it, exiting from the follower's line of sight before the follower had a chance to react. This forced the follower agent to switch behavior, and re-locate its target.

Figure 7 shows 10 additional trials, this time when an explicit maintenance condition was put in place. Here, we added an individual task-maintenance condition to the recipe of the follower, instructing it to keep the leader in focus while moving. The figure shows that now, the follower agent no longer loses track of the leader, since it actively pans to track the leader. This is an example of how task related goals can be set apart from maintenance related goals, adding new flexibility to behavior-based architecture and clarity to the code: It was achieved without changing the *explore-movement* or *move-to-target* behaviors, allowing to keep them simple and compact.

## 5.2 Collaborative vs. individual maintenance

The results in the previous section show the importance of maintenance during behavior execution. However, one could point out that no teamwork is really being tested in these scenarios since no communication or coordination takes place. Thus perhaps the improvements we are seeing in transition from only carrying out achievement actions, to using the maintenance mechanism, are limited to the individual. In other words, is there really any benefit to collaborative maintenance, compared to individual maintenance?

This time, we use BITE to answer this question. We have created two versions of a BITE behavior graph, implementing a diamond-

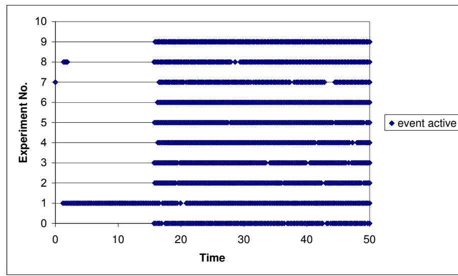



**Figure 7: Maintenance of the *see-leader* event by the follower.**

shaped formation for four simulated robots. In the collaborative maintenance version (Figure 5(a)) the simulated robots use collaborative maintenance, whereby the leader takes responsibility for maintaining the distance and angle to its followers, and communicates its movements so that they can calculate their own new positions. In the individual maintenance version (Figure 5(b)), the followers use maintenance behaviors to visually keep track of the leader's position, but the leader is not responsible for this distance.

We conducted multiple trials using BITE in both individual and collaborative maintenance configurations. We set up five obstacle courses, marked A through E. In course A the simulated robots moved straight, but a long fence, parallel to the movement direction, separated the right-most robot from the leader. In course B, the leader took a sharp turn that caused it to be blocked from the view of the rear follower (it was occluded by the leftmost robot). Course C was a repeat of course A, but here the fence was missing portions in regular intervals (forming a kind of dashed fence). These caused the rightmost robot to repeatedly lose sight of the leader, and then catch up with it again. Course D consisted of simple movement forward with no obstacles. Course E consisted of a short segment forward, then a 90-degree turn to the left, another short segment, and then a 90-degree turn to the right (all of this with no obstacles).

Each course was repeated 5–10 times in each configuration (collaborative, individual). We measured the time to complete the course, and the average error in maintaining the formation. This error was calculated by examining the distance between the actual position of each simulated robot, and the position it should have ideally maintained given the position of the leader.

Figures 8 and 9 show the results from these experiments. In both figures, the dark column shows the results of using collaborative maintenance, and the light column shows the results of using individual maintenance. The vertical lines on the bars mark standard deviations. The figures show that in all courses, the use of collaborative maintenance leads to significantly improved results (see below for a discussion of courses A and B). All results were found to be significant using a one-tailed t-test, except for the difference in time in course E, where no significant difference was found.

In courses A and B, the individual maintenance versions of the task could not complete the course, and so these runs had to be stopped. Nevertheless, we measured the positional error until the point in which the simulated robots were stopped. This lead to the seemingly contradictory result that in course B, the positional error was lower with individual maintenance than with collaborative maintenance. This was because course B consisted of a very sharp turn in which necessarily positional errors increase. Since the individual maintenance version were stuck before the sharp turn, their position error appeared smaller.

We stress the difference between individual and collaborative maintenance goals using another experiment with DIESEL . Here,

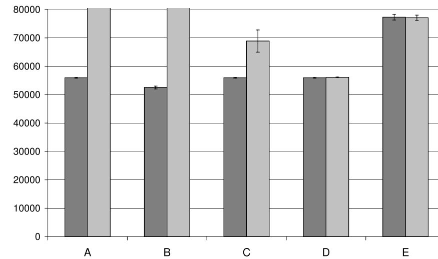



**Figure 8: Results from the BITE experiments: Maximum time in courses A and B indicates that the experiment had to be stopped for lack of progress.**

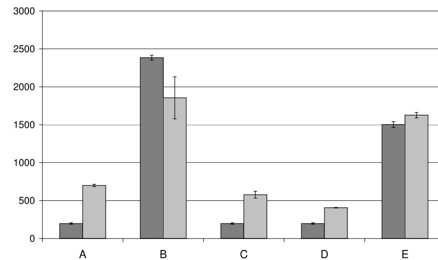



**Figure 9: Results from the BITE experiments: Position Error.**

we chose a square-shaped corridor, in which the leader could run indefinitely. With every turn, the leader could potentially be blocked from the view of the follower, and thus the agents had many opportunities to lose each other. Using individual maintenance, the leader would not be responsible for maintenance of the distance to the follower, and it would be up to the follower to carry out all actions necessary to maintain the distance. In collaborative maintenance, both leader and follower share the burden for maintaining the goals of the team.

To see this, we manually introduced a failure into the scenario above, where the follower was physically blocked from moving forward. While the follower agent proactively seeks to maintain the presence of see-leader events, the leading agent uses reactive maintenance, meaning it acts only when such an event drops. In this failure case, once the follower stopped tracking the leader, the leader's positive-maintenance is proposed (even while it was heading to its designated target), and the leader waits.

Figure 10 shows the results of such a case. The figure shows on the X-axis the passage of time (in Unreal Tournament seconds). The Y-axis shows the distance between the follower and leader. With individual maintenance, the distance between leader and follower continue to grow after the failure occurs. However, with team maintenance, distance between both agents is kept throughout the artificially-introduced failure.

## 5.3 Teamwork Maintenance

The previous sections have evaluated the use of maintenance in the context of task behaviors. One novelty in the mechanism we introduced is that it can be re-used for maintaining the team hierarchy in face of catastrophic failures to individual agents. We call this teamwork-maintenance, to contrast with task-maintenance described in the previous sections.

To demonstrate team-maintenance in DIESEL , we divided four agents into two groups, each consisting of a leader and a follower. We defined a single team-maintenance condition in each team, stating that each agent should have a coordinator at any given moment.

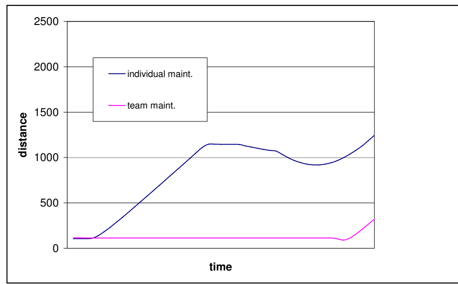



**Figure 10: Distance between leader and follower, in cases of individual and team goal maintenance.**

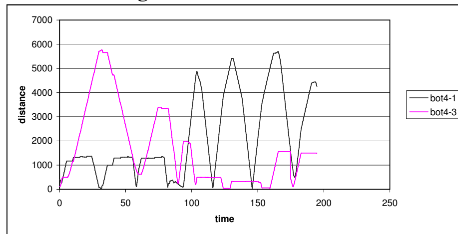



**Figure 11: Maintenance of team hierarchy: Distance between bot4 and bot3, bot1.**

In each team, the coordinator was initially set to be the leading agent. In team A, consisting of bot1 and bot2, it was bot1, and in team B, consisting of bot3 and bot4, it was bot3. This was part of the team-hierarchy for each agent. Both teams followed the same recipe previously described, with the two leaders independently leading their respective followers in constant movement along the corridor.

To show teamwork maintenance in action, we deliberately blocked any contact with bot3 and hid it during the first half of the experiment. As a result, bot4, changed its coordinator, and began following bot1, by joining team A. After running half of the experiment in such a manner, we removed the blocking on the original coordinator, bot3, thus allowing bot4 to fall back to its original team, team B. Figure 11 shows the distance between bot4 and bot1, and between bot4 and bot3. The figure shows how in the distance between bot4 and bot3 (the hidden leader) was greater than the distance between bot4 and bot1 (the alternate leader). The situation is reversed once bot3 is seen again, and bot4 switches back to its original leader.

Switching teams in this example is achieved by a team-maintenance behavior (operator, in Soar), which manipulates bot4's team-hierarchy. The behavior works by checking whether at any given time a coordinator is unreachable. If so, then the behavior finds a new team in which there is a team coordinator and change the organizational membership of the agent to be a part of the other team. Since this is only a maintenance behavior, as opposed to a regular one, if the exception is resolved, the maintenance behavior is terminated, and regular order is restored.

# 6. CONCLUSIONS AND FUTURE WORK

This paper argued for the introduction of a general mechanism for collaborative goal maintenance in teamwork architectures. We presented such a mechanism, and described its integration within two implemented architectures for teamwork: DIESEL , an architecture built on top of the Soar cognitive architecture [13]; and BITE , an architecture for controlling teams of behavior-based robots. We empirically demonstrated that the use of proactive maintenance leads to improved performance compared to reliance on achievement actions (also used as a reactive form of goal maintenance). We also showed that the use of collaborative maintenance, in which all team-members take responsibility for maintaining the team goals, leads to improved results compared to individual maintenance. Finally, we showed how the maintenance mechanism can be used to maintain the team structure. This allows the programmer to focus more clearly on achievement and maintenance aspects of the task, and to separate completely the issue of how to maintain the team-structure in face of catastrophic failures. Future work includes exploring a diverse set of maintenance protocols for taskwork and teamwork.

# 7. REFERENCES

[1] T. Balch and R. C. Arkin. Behavior-based formation control for multi-robot teams. *IEEE Transactions on Robotics and Automation*, 12 1998.

[2] P. R. Cohen and H. J. Levesque. Teamwork. *Nous*, 35, 1991.

[3] B. P. Gerkey, R. T. Vaughan, and A. Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the International Conference on Advanced Robotics*, pages 317–323, 2003.

[4] B. J. Grosz and S. Kraus. Collaborative plans for complex group actions. *AIJ*, 86:269–358, 1996.

[5] M. J. Huber. JAM: A BDI-theoretic mobile agent architecture. In *Agents-99*, pages 236–243, 1999.

[6] N. R. Jennings. Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. *AIJ*, 75(2):195–240, 1995.

[7] G. A. Kaminka and I. Frenkel. Flexible teamwork in behavior-based robots. In *AAAI-05*, 2005.

[8] G. A. Kaminka, M. M. Veloso, S. Schaffer, C. Sollitto, R. Adobbati, A. N. Marshall, A. Scholer, and S. Tejada. GameBots: A flexible test bed for multiagent team research. *Communications of the ACM*, 45(1):43–45, January 2002.

[9] S. Kumar and P. R. Cohen. Towards a fault-tolerant multi-agent system architecture. In *Agents-00*, pages 459–466, Barcelona, Spain, 2000. ACM Press.

[10] S. Kumar, P. R. Cohen, and H. J. Levesque. The adaptive agent architecture: Achieving fault-tolerance using persistent broker teams. In *ICMAS-00*, pages 159–166, Boston, MA, 2000. IEEE Computer Society.

[11] J. Lee, M. J. Huber, P. G. Kenny, and E. H. Durfee. UM-PRS: An implementation of the procedural reasoning system for multirobot applications. In *Proceedings of the Conference on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS-94)*, pages 842–849, 1994.

[12] N. Lesh, C. Rich, and C. L. Sidner. Using plan recognition in human-computer collaboration. In *UM-99*, Banff, Canada, 1999.

[13] A. Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, Massachusetts, 1990.

[14] L. E. Parker. ALLIANCE: An architecture for fault tolerant multirobot cooperation. *IEEE Transactions on Robotics and Automation*, 14(2):220–240, April 1998.

[15] M. Tambe. Towards flexible teamwork. *JAIR*, 7:83–124, 1997.

[16] M. Tambe and W. Zhang. Towards flexible teamwork in peristent teams. In *ICMAS-98*, 1998.

[17] J. Yen, J. Yin, T. R. Ioerger, M. S. Miller, D. Xu, and R. A. Volz. CAST: Collaborative agents for simulating teamwork. In *IJCAI-01*, pages 1135–1144, 2001.