DEVELOPMENT OF TEAMWORK IN PHYSICAL AGENTS

Yakir Ari
yakirari@cs.biu.ac.il

September 2007

Bar-Ilan University
Department of Computer Science

DEVELOPMENT OF TEAMWORK IN PHYSICAL AGENTS

by

Yakir Ari

Advisor: Dr. Gal Kaminka

Submitted in partial fulfillment of the requirements for the Master's degree
in the department of Computer Science

Ramat-Gan, Israel
September 2007

This work was carried out under the supervision of

Dr. Gal A. Kaminka

Department of Computer Science, Bar-Ilan University.

Acknowledgments

I would like to thank my advisor Dr. Gal A. Kaminka that followed me every step of the way and pushed me on when I needed. Supported my research with his vast knowledge and helped me formulate my ideas into coherent research. Gal is a great research partner, an amazing friend, a superb mentor and was always a pleasure to work with.

I feel deep gratitude to my MAVERICK colleagues for their contribution in excellent advices, insightful ideas and unfailing enthusiasm. Among them, I would like to thank Dan Erusalimchick for his contribution to the BITE'M experiments, building the first running version of BITE with Maintenance, and Nirom Choen Nov, for being my co-author on the Maintenance paper.

We thank our Elbit Systems partners for many useful discussions and feedback: Ora Arbel, Itay Guy, Ilana Segall, Myriam Flohr, and Erez Nachmani. The work was supported in part by a generous gift by MAK International Confer-Technologies, and by BSF grant #2002401.

I want to thank my parents Yochi and Doron Yakir for listening to my lectures, presentations and practice talks all along the way. I think their knowledge in the field is by now substantial.

Last but not least Meytal my love and wife, who was always beside me and not only tolerated the long hours at the lab but even provided snacks.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction to the thesis

This thesis explores novel challenges in development execution of team oriented programs in dynamic, complex domains. Examples of such domains include: MAK Virtual Forces, a platform used in Simulation and Training applications; USARSim, Gamebots and SoarBots in Unreal Tournament used for the simulation of various types of agents, and for the simulation of physical Robots in the rescue domain; and Player/Stage environment for the simulation of various robots including pioneer and rv400 autonomous vacuum cleaners.

The first part if this thesis introduces collaborative maintenance goals to teamwork architectures. We propose collaborative task and team maintenance as an innovative mechanism, alternative to sequences of goals achievement and individual maintenance. We show its benefits in comparison to several alternative methods, and provide results from the various domains.

The second part of the thesis describes the pre-deployment tools of development for multi-agent teams. We dive into our development environment (IDE), showing state-of-the-art facilities such as refactoring and testing for agent applications. Our IDE is object-oriented, facilitating coding by the use of pre-made templates, re-usability of components such as plans and behaviors, instead of wizzards and graphical means of programming.

Both parts of this thesis where published, the first "Towards Collaborative Task and Team Maintenance" was presented in AAMAS07 [20] , and the second "An Integrated Development Environment and Architecture for Soar-Based Agents" was presented in IAAI07 [38].

Several additionl papers which use the recipe mechanism presented in this thesis for the DIESEL Architecture were published: "Social comparison for modeling crowd behavior" by Fridman & Kaminka AAMAS 2007 [16] and "Computational Load and Performance in Integrated Multi-Agent Intention recognition" by Nirom Cohen-Nov & Gal A. Kaminka presented at BISFAI07.

Chapter 2

Towards Collaborative Task and Team Maintenance

2.1 Chapter 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 chapter presents a 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'M , 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.

2.2 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 [32], robotics [29, 14], industrial distributed systems [12], and collaborative user interface [25].

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 [36].

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 [22, 21] and continual task allocation [29]. 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, often used in theoretical studies of teamwork (e.g., [4]). Existing teamwork architectures, based on teamwork theory [4]), 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 2.3), existing teamwork theory and teamwork architectures do not account for such communications.

We 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 re-use 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 top of Soar [28] and used in virtual worlds, and BITE'M , an extended version of BITE [14, 15], 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.

2.3 Related Work

Duff et al. [5] have recently proposed a model of proactive goal maintenance for BDI agents, similar to the one we present in this work. Their work focuses on extending the BDI architecture for a single agent. In contrast, the model we propose allows modeling of *collaborative goals*, modeling the joint responsibility of teammates to proactively or reactively maintaining a condition; Thus our work is a more general case. Moreover, we show how to utilize such collaborative

maintenance to also address maintenance of team organization, rather than only task-related conditions.

Focusing on teamwork architectures, we note that most have only allowed for achievement goals. We therefore focus here only on those that have addressed maintenance goals to some extent.

Kumar and Cohen [22, 21] 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 [22, 21], 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'M 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 [32], implemented in Soar [28], focuses for the most part on achievement goals. However, a first step towards extending STEAM towards maintenance goals was introduced in [36]. 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 thesis, deals with PTPM teams, i.e., persistence of team structure. We refer to this as teamwork maintenance. However, in contrast to [36], DIESEL and BITE'M 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 [39] 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 [29] 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.

2.4 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. The architecture extends structures common to situated agent architectures, and the algorithms used with these structures.

We begin by taking a brief look at the structures and algorithms of existing teamwork architectures, that do not support maintenance (Section 2.4.1). We then show how these are extended to support maintenance (Section 2.4.2). These extensions require significant changes to the underlying control loop of the agents, and to the parallelism it must support.

2.4.1 Existing Situated Agent Teamwork Architectures

We begin by a brief overview of situated agent control, upon which the maintenance mechanism is based. Modern Belief-Desire-Intention (BDI) and behavior-based control architectures utilize a connected, directed graph, that defines a library of behaviors or actions by which agents achieve their goals. Nodes in the graph denote atomic or complete actions (behaviors), and edges signify decomposition or temporal relations between them. A control algorithm selects actions for execution, based on the currently executing behaviors, and the world state.

Structures

We follow [14, 15] in formally defining a *task behavior graph*, as an augmented connected graph $\langle B, S, V, b_0 \rangle$, where:

- B is a set of vertices. Each vertex in B is a goal-achieving controller, called a *behavior* (in Soar, *operator*). Each behavior has preconditions which enable its selection (the agent can select between enabled behaviors), and termination conditions that determine when its execution must be stopped (if previously selected).

- S and V are sets of directed edges between behaviors ($S \cap V = \emptyset$).

- S is a set of *sequential* edges, which specify temporal order of execution of behaviors. A sequential edge from b_1 to b_2 specifies that b_1 must be executed before executing b_2. A path along sequential edges, i.e., a valid sequence of behaviors, is called an *execution chain*.

- V is a set of vertical *task-decomposition* edges, which specify how a controller can be decomposed into execution chains containing multiple lower-level behaviors. Sequential edges may form circles, but vertical edges cannot. Thus behaviors can be repeated by choice, but cannot be their own ancestors.

- $b_0 \in B$ is a behavior in which execution begins.

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).

This type of structure, possibly with some minor variations, appears as the basis for many situated agent architectures (e.g., [24, 11, 28]). It is referred to as a *recipe* [9], a *plan hierarchy* [32], or *behavior graph* [14, 15]. We will use these terms interchangeably.

An example recipe is shown in Figure 2.1. 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 at a given time.

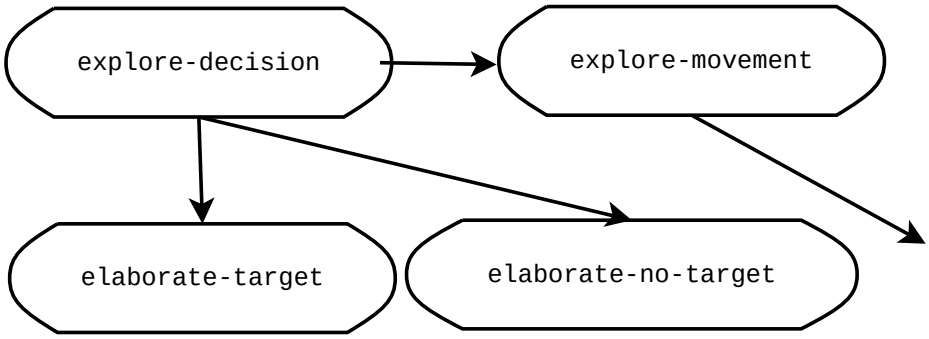


Figure 2.1: **An example recipe.**

To maintain knowledge of the organizational structure of the team, a second structure—*team hierarchy*—is often used in architectures that support situated teamwork, such as TEAMCORE [35], MONAD [6] and BITE [14, 15]. 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 ([12, 32]). We focus here on a brief description of the team hierarchy structure itself.

A *team hierarchy* is a DAG (Directed Acyclic Graph) whose vertices are associated with sub-teams of agents, and whose edges signify sub-team-membership relationships [35, 6]. Several vertices appear in any organization hierarchy: Given the complete set of robot team-members R, a vertex corresponding to R (and representing the entire organization) is a part of the hierarchy, as are all the singleton sets $\{r_i\}$, where $r_i \in R$. Other vertices correspond to multi-robot sub-teams of robots in R and are connected such that if there exists an edge $\langle R_1, R_2 \rangle$, then $R_2 \subset R_1$. The team hierarchy thus forms a partial lattice, from the root team R which includes all team-members, to sub-teams corresponding to each of the members by itself (i.e., to the individuals in the organization). To allow behaviors to reason about the organizational unit responsible for their execution (and vice-versa), BITE uses bi-directional links between the behavior graph and the team hierarchy. A link from a behavior B_j points to a sub-team R_i (and back) if B_j is to be jointly executed by R_i.

We note in passing that BITE maintains a third structure, holding a set of *social interaction behaviors* which control inter-agent interactions. Interaction behaviors typically control communications and execute protocols (e.g., voting) that govern coordinated activity. For instance, a **synchronized selection** algorithm is triggered when new team behaviors are selected for execution, in particular when a decision is to be made between several sequential transitions. See [14, 15] for additional details.

Control Algorithm for Goal-Achieving Agents

In [14, 15], each of the robots executes Algorithm 1, using its own copy of three structures, a behavior graph $\langle B, S, V, b_0 \rangle$, a team hierarchy T and an interaction behaviors set O[1]. The control loop executes *a behavior stack*—root behavior to

[1]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

leaf—where top behaviors on the stack are executed simultaneously with their currently selected children.

Execution begins by pushing the initial behavior of the graph on the execution stack (lines 1–2). Then the algorithm loops over four phases in order:

1. It recursively expands the children of the behavior, allocating them to sub-teams if necessary (lines 3a–3c).

2. It then executes the behavior stack in parallel, waiting for the first behavior to announce termination (lines 4a–4c). All descendants of a terminating behavior are popped off the stack (i.e., their execution is also terminated—line 4b).

3. A synchronized termination takes place (line 6). This can result in a newly-allocated behavior within the current parent context, in which case, it will be put on the stack for expansion (line 7).

4. Otherwise, this indicates that the robot should select between any enabled sequential transitions from the terminated behavior (lines 8a–8e). This process normally results in new behaviors put on the stack, and then a final goto (line 9) back to line 3 begins again.

The recursive allocation of children behaviors to sub-teams in lines 3a–3c relies on the call to the $Allocate()$ procedure. It takes the current execution context (i.e., current stack, available children), and then calls the appropriate social interaction behavior in O (linked from the current parent) to make the allocation decision. The current execution stack is used to help guide allocations, e.g., by conveying information about where in the behavior graph the allocation is taking place. In addition, the interaction behavior is given access to any links from the parent behavior to the team hierarchy, e.g., to determine whether any children task-behaviors are already pre-allocated. Once a final allocation is determined, $Allocate()$ is responsible for updating the links from the behavior graph to the team hierarchy (and vice versa) to reflect the allocation. It then returns, for each

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 *interaction behaviors*.

Algorithm 1 Control

Input: behavior graph $\langle B, S, V, b_0 \rangle$, team hierarchy T, interaction behaviors set O

1. $s_0 \leftarrow b_0$ // initial behavior for execution
2. push s_0 onto a new behavior stack G.
3. while s_0 is non-atomic // has children
   (a) $A \leftarrow \{b_i\}$, s.t., $\langle s_0, b_i \rangle$ is a decomposition edge
   (b) if A has only one behavior b, $push(G, b)$.
   (c) else $b \leftarrow$*Allocate*(G, s_0, A, T, O), $push(G, b)$.
   (d) $s_0 \leftarrow b$.
4. execute in parallel for all behaviors b_i on G: // Execution
   (a) execute b_i until it terminates
   (b) while $b_i \neq top(G)$, $pop(G)$
   (c) break parallel execution, goto 5.
5. $b \leftarrow pop(G)$ // Terminate joint execution
6. $c \leftarrow$*Terminate*(G, b, T, O)
7. if $c \neq NIL,\ push(G, c)$
8. else: // Select next behavior in execution chain
   (a) Let $Q \leftarrow \{s_i\}$, s.t. $\langle b_0, s_i \rangle$ is a sequential edge
   (b) if Q is empty, goto 5 // terminate parent
   (c) if Q has one element s, $push(G, s)$
   (d) else $s \leftarrow$*Decide*(G, b_0, Q, T, O)
   (e) $s_0 \leftarrow s$
9. If G not empty, goto 3.

robot, the child behavior for which it is responsible as part of the split sub-team (or individually, if the sub-team is composed only of the individual robot).

Synchronized termination (line 5–7) and selection (lines 8a–8e) similarly rely on calls to the procedures $Terminate()$ and $Decide()$, respectively. $Terminate()$ is responsible for evoking the execution termination interaction behavior, which can return a new child behavior for execution under the current parent. If it doesn't, then the next behavior in the execution chain must be selected by $Decide()$, which calls a synchronization interaction behavior. Since synchronized selection involves all members of the current sub-teams selecting together, this behavior would normally communicate with the members of the sub-team assigned to the terminated behavior. Note that in step 8b we also handle the case where no more behaviors are available in the execution chain. This case signals a termination of an execution chain, which in turn signals termination of the parent, thus the branching back to line 5.

2.4.2 Collaborative Maintenance Behaviors

To allow situated agent architectures to work with maintenance goals, rather than only achievement goals, several extensions are required to their structures and algorithms. These are described below.

Extending the Behavior Graph

To represent conditions to be maintained during execution, 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.

Maintenance Behaviors in a Behavior Graph. This association of maintenance behaviors to maintenance conditions on goal-achieving behaviors essentially adds a third type of edges to the behavior graph. In addition to the sequential and decomposition edges previously discussed, a behavior graph now includes *maintenance edges*, which connect maintenance behaviors to goal-achieving behaviors.

Formally, the *extended* task behavior graph is an augmented connected graph tuple $\langle B, M, S, V, MV, b_0 \rangle$, where:

- B is a set of vertices representing goal-achieving behaviors (as before).
- M is a set of vertices representing maintenance behaviors, where $B \cap M = \emptyset$.
- S, V and MV are non-intersecting sets of directed edges between behaviors. S, V have been previously discussed. Edges in MV always take the form $\langle b, m \rangle$, where $b \in B$, and $m \in M$. The existence of an edge $\langle b, m \rangle \in MV$, from a behavior b to a maintenance behavior m implies that m may be selected to maintain a maintenance condition on b.
- $b_0 \in B$ is a behavior in which execution begins.

The existence of several edges $\langle b, m_i \rangle$, where $m_i \in M$ and $i > 1$ implies that *all* of the maintenance behaviors m_i may be executed. To force selection between alternatives, a single maintenance behavior m_1 can be decomposed into sub-behaviors, as is done with the goal-achieving behaviors previously discussed. Indeed, all the usual semantics of the different edge types, and the constraints on their use within the graph, remain in effect. Notice that we explicitly prohibit

maintenance behaviors from having maintenance behaviors linked to them, as in practice we found this to be of little use.

Though the use of maintenance conditions in integrated architectures is rare, the key novelty described in this thesis is the ability to tie specific *team behaviors* to these conditions. The behaviors will be triggered automatically by DIESEL or BITE'M , 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.

Maintenance of the Team Hierarchy. 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.

Control Algorithm with Maintenance

The control algorithm using the extended task behavior graph needs to be extended as well, to execute the maintenance behaviors *in parallel* to any goal-achieving behaviors[2]. As with BITE , each of the agents in the team executes its own copy of the control algorithm, and its own copy of the extended task behavior graph.

Algorithm 2 accepts an extended task behavior graph $\langle B, M, S, V, MV, b_0 \rangle$, a team hierarchy T, an interaction behaviors set O, and an execution stack G. As with Algorithm 1, the control loop executes *a behavior stack*—root behavior to leaf—where top behaviors on the stack are executed simultaneously with their currently selected children.

However, unlike Algorithm 1, Algorithm 2 recursively creates new execution stacks for maintenance behaviors, and thus runs multiple stacks of executing behaviors in parallel. One way to think about the difference between the two algorithms is this: While Algorithm 1 maintains a stack of running threads, Algorithm 2 maintains a thread tree, where threads can spawn (or fork) several children threads in parallel; each path of decomposition edges acts as a stack.

Execution begins by pushing the initial behavior of the graph on the execution stack (lines 1–2). Then the algorithm loops over four phases in order:

1. It recursively expands and allocates the children of a behavior. While doing so, parallel control loops are being initiated for all maintenance behaviors (lines 3a–3f).

2. It then executes the behavior stack in parallel, waiting for the first behavior to announce termination (lines 4a–4c). All descendants of a terminating behavior are popped off the stack (i.e., their execution, and the maintenance control loop, is also terminated—line 4b).

3. A synchronized termination takes place (line 6). This can result in a newly-allocated behavior within the current parent context, in which case, it will be put on the stack for expansion (line 7).

[2]We thank Dan Erusalimchik for making this observation, and for implementing the BITE'M version of this algorithm

Algorithm 2 Control with Maintenance

Input: behavior graph $\langle B, M, S, V, MV, b_0 \rangle$, team hierarchy T, interaction behaviors set O and execution stack G

1. $s_0 \leftarrow b_0$ // initial behavior for execution
2. push s_0 onto stack G.
3. while s_0 is non-atomic // has children or maintenance behaviors
   (a) $A \leftarrow \{b_i\}$, s.t., $\langle s_0, b_i \rangle$ is a decomposition edge $\in V$
   (b) $G' \leftarrow \emptyset$
   (c) forall $\{m_i | \langle s_0, m_i \rangle \in MV\}$
      i. create new execution stack g'
      ii. $G' \leftarrow G' \cup g'$
      iii. Recursively call **Control with Maintenance** with the inputs:
         - behavior graph $\langle B, M, S, V, MV, m_i \rangle$
         - team hierarchy T
         - interaction behaviors set O
         - execution stack g'
   (d) if A has only one behavior b, $push(G, \langle b, G' \rangle)$.
   (e) else $b \leftarrow$*Allocate*(G, s_0, A, T, O), $push(G, \langle b, G' \rangle)$.
   (f) $s_0 \leftarrow b$.
4. execute in parallel for all $\langle b_i, G' \rangle$ on G: // Execution
   (a) execute b_i until it terminates
   (b) while $b_i \neq top(G)$, $pop(G)$
   (c) break parallel execution, goto 5.
5. $b \leftarrow pop(G)$ // Terminate joint execution
6. $c \leftarrow$*Terminate*(G, b, T, O)
7. if $c \neq NIL,\ push(G, c)$
8. else: // Select next behavior in execution chain
   (a) Let $Q \leftarrow \{s_i\}$, s.t. $\langle b_0, s_i \rangle$ is a sequential edge
   (b) if Q is empty, goto 5 // terminate parent
   (c) if Q has one element s, $push(G, s)$
   (d) else $s \leftarrow$*Decide*(G, b_0, Q, T, O)
   (e) $s_0 \leftarrow s$
9. If G not empty, goto 3.

4. Otherwise, this indicates that the robot should select between any enabled sequential transitions from the terminated behavior (lines 8a–8e). This process normally results in new behaviors put on the stack, and then a final goto (line 9) back to line 3 begins again.

The recursive allocation of children behaviors to sub-teams in lines 3a–3f relies on the call to the $Allocate()$ procedure. As with Algorithm 1, it takes the current execution context (i.e., current stack, available children), and then calls the appropriate social interaction behavior in O (linked from the current parent) to make the allocation decision. The current execution stack is used to help guide allocations, e.g., by conveying information about where in the behavior graph the allocation is taking place.

However, in step 3c, a significant departure from Algorithm 1 occurs, as new execution stacks are created—and execution thereof is initiated—for maintenance behaviors. This effectively triggers independent, parallel, execution of these maintenance behaviors on the g' stack. In order to associate these stacks with the goal-achievement behavior that triggered them, they are collected together in the set G', and are associated with the behavior as it is pushed on the stack G.

2.5 Two Implementations

We have implemented the maintenance mechanism described above in two different architectures. DIESEL , built on top of the Soar integrated cognitive architecture [28], was built from the ground up with collaborative maintenance in mind. The other implementation revisited the BITE behavior-based multi-robot architecture [14, 15], and extended it to support maintenance behaviors, creating a new architecture called BITE'M (BITE with Maintenance). The two architectures were used in very different settings, and this serves as evidence of the generality of the mechanism: It was successfully integrated in both. We describe the implementations below.

2.5.1 DIESEL

We use Soar for the implementation of our architecture. While a comprehensive description of Soar is beyond the scope of this section, we provide a brief overview here. Soar is a general cognitive architecture for developing systems that exhibit intelligent behavior [28, 31]. 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 hierarchy, 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'M 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 [18], an adversarial game environment that enables qualitative comparison of different control techniques. Figure 2.2 shows a screen-shot of Soar agents in the GameBots domain, running DIESEL .

Figure 2.2: **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.**

2.5.2 BITE'M

BITE (Bar Ilan Teamwork Engine) 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 [14] (Figure 2.3). The version we use here, called BITE'M (BITE with Maintenance), was extended with maintenance capabilities as described above. It is implemented for the player-stage system, a de-facto standard API for controlling different types of physical and simulated robots [8], rather than only AIBOs; code running in the simulation

Figure 2.3: **Sony AIBO robots moving in formation, controlled by** BITE **. Taken from [14, 15]**

can be used with few modifications on the physical robots. Figure 2.4 shows a view of BITE'M -controlled robots moving in formation, in the player-stage simulator.

The use of BITE (as opposed to the newer BITE'M) 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 distance and angle to the leader (see, for instance [1]). In 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'M , 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'M

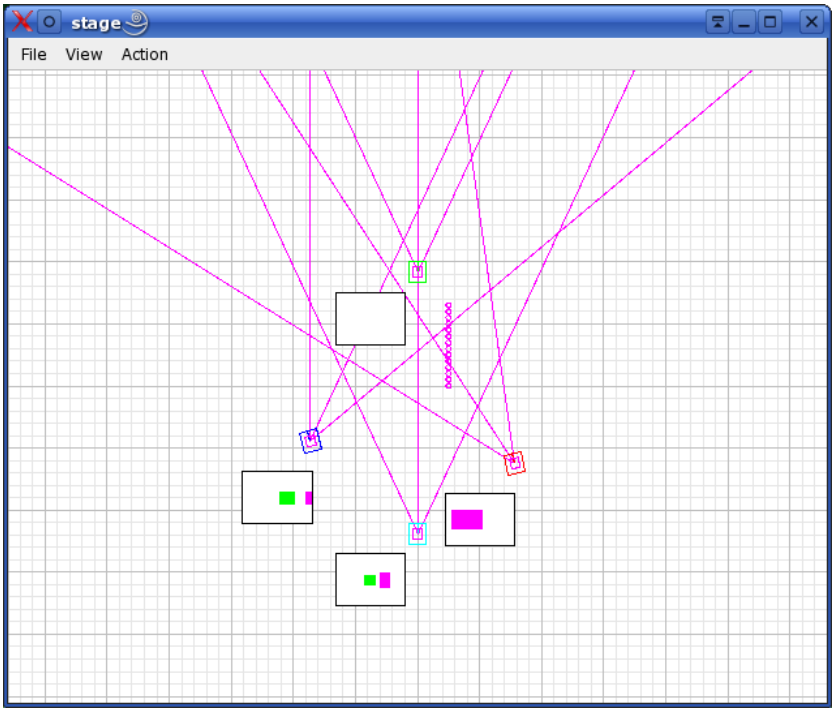
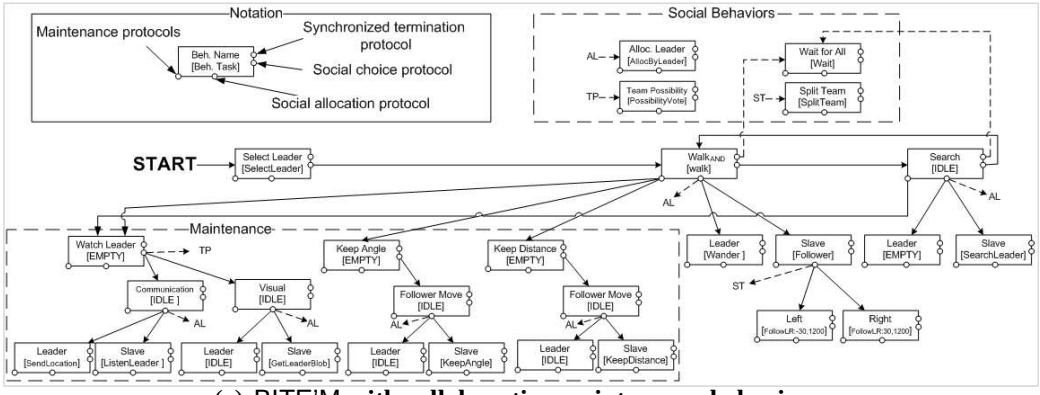


Figure 2.4: **Robots running BITE'M in the player-stage API. The robots form a diamond. The lines mark visual field of view. Boxes with filled blocks show the colors perceived by each robot.**

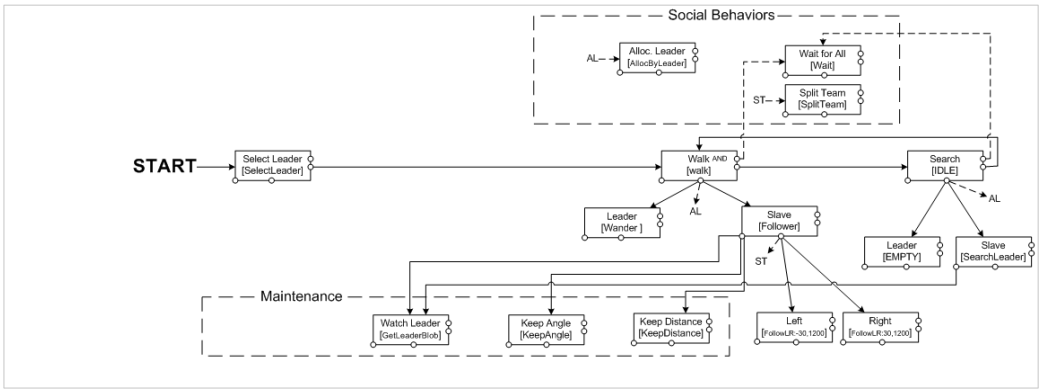
automatically have the leader communicate its position or take other actions, such that its followers can track it more easily. Moreover, the implementation in BITE allows to easily see the difference between individual and collaborative maintenance.

Figure 2.5 shows the behavior graph, social behaviors, and maintenance behaviors in BITE'M , for the formation-maintenance task. Figure 2.5(a) shows the behavior structures when using the collaborative maintenance mechanism. Figure 2.5(b) shows the same task, but with individual maintenance. The upper-left corner of Figure 2.5(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'M triggers automated coordination (see [14] for details, which are outside the scope of this thesis).

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



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



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

Figure 2.5: **Collaborative and individual maintenance behaviors in** BITE **.**

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 2.5(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 2.5(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 2.5(a) as compared to 2.5(b).

2.6 Evaluation

To evaluate the contribution offered by the introduction of collaborative maintenance in DIESEL and BITE'M , 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 hypotheses. A first set of experiments (Section 2.6.1) 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 2.6.2) 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 2.6.3) demonstrates how team reconfiguration occurs using the maintenance mechanism with the team hierarchy, rather than the behavior graph.

2.6.1 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 [18]. In all experiments, we used the same recipe (Figure 2.1), 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 2.1. 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 in-

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

Table 2.1: Individual achievement (reactive maintenance) compared to individual proactive maintenance.

deed 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 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 2.6 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 2.7 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

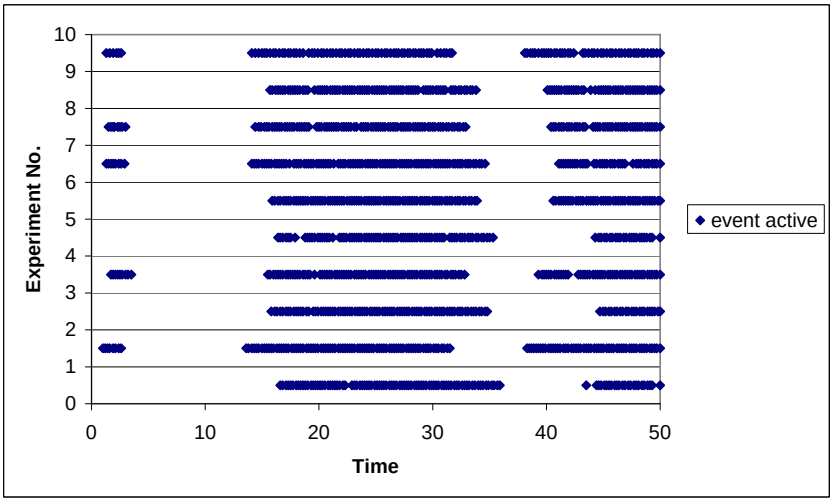


Figure 2.6: ***see-leader* event logged by the follower agent. No maintenance conditions.**

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.

2.6.2 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'M to answer this question. We have created two versions of a BITE'M behavior graph, implementing a diamond-shaped formation for four simulated robots. In the collaborative maintenance version (Figure 2.5(a)) the simulated robots use collaborative maintenance, whereby the leader takes respon-

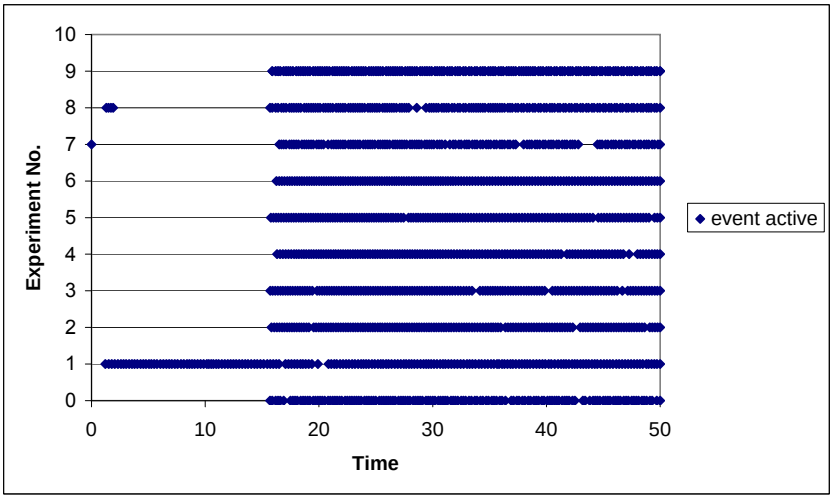


Figure 2.7: **Maintenance of the *see-leader* event by the follower.**

sibility 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 2.5(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'M 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

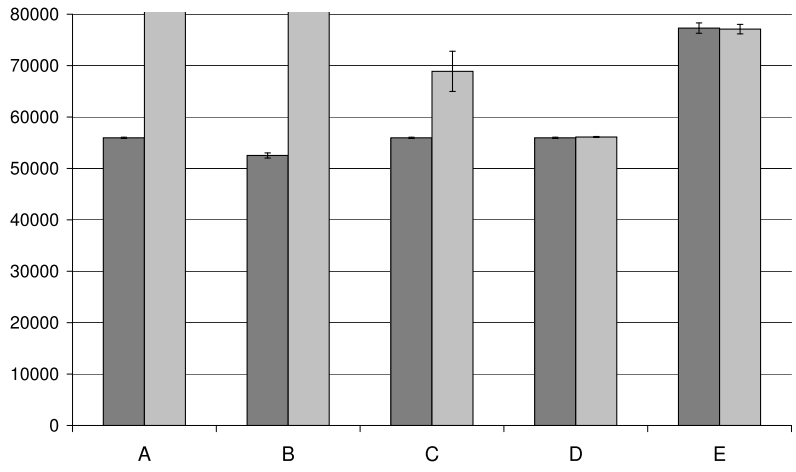


Figure 2.8: **Results from the BITE experiments: Maximum time in courses A and B indicates that the experiment had to be stopped for lack of progress.**

between the actual position of each simulated robot, and the position it should have ideally maintained given the position of the leader.

Figures 2.8 and 2.9 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

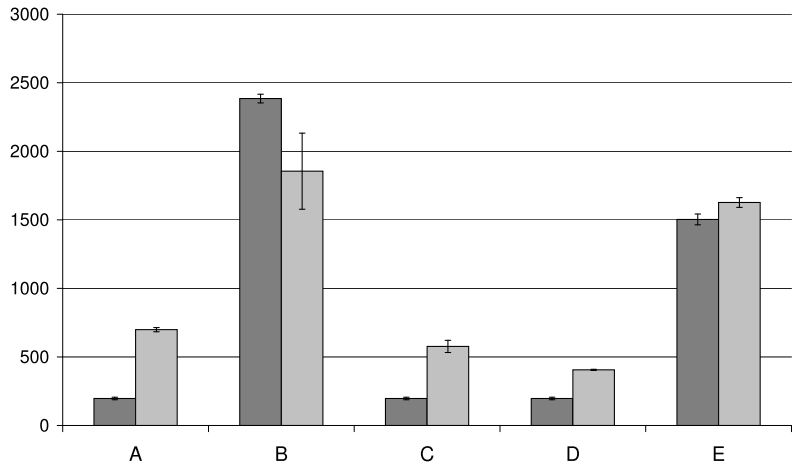


Figure 2.9: **Results from the BITE experiments: Position Error.**

goals using another experiment with DIESEL . Here, 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 2.10 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

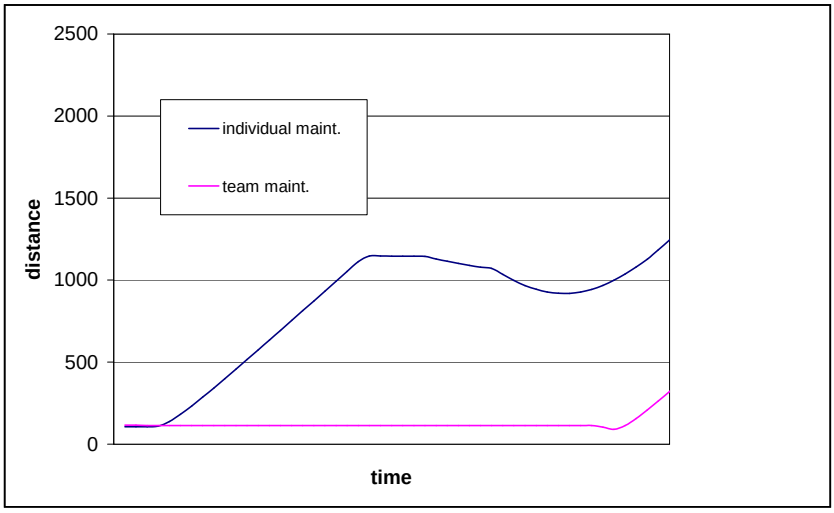


Figure 2.10: **Distance between leader and follower, in cases of individual and team goal maintenance.**

artificially-introduced failure.

2.6.3 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.

In the Unreal Tournament domain, 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. 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 con-

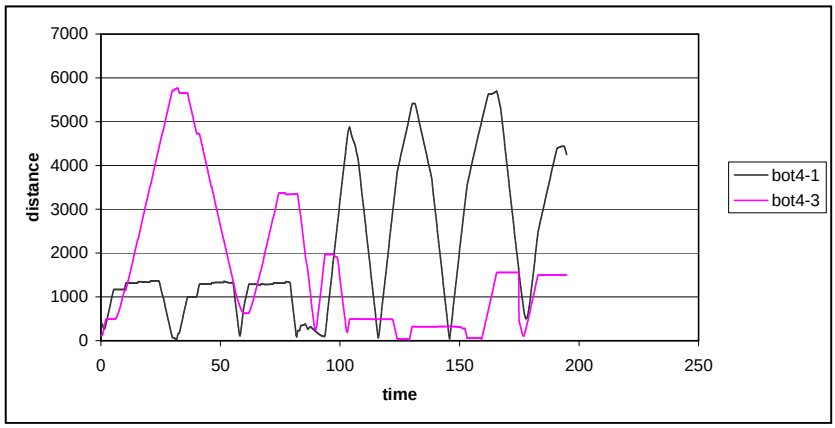


Figure 2.11: **Maintenance of team hierarchy: Distance between bot4 and bot3, bot1.**

tact 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 2.11 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.

Another evaluation of Teamwork Maintenance, was achieved by the collaboration with Elbit Systems, Ltd. on the Smart Entities project. Elbit built several enhancements based on the MAK simulation, called VRF(Virtual Forces), aiming for better Simulation and Training applications. DIESEL was adopted as a mechanism for simulating synthetic teams executing elaborated plans, while reacting

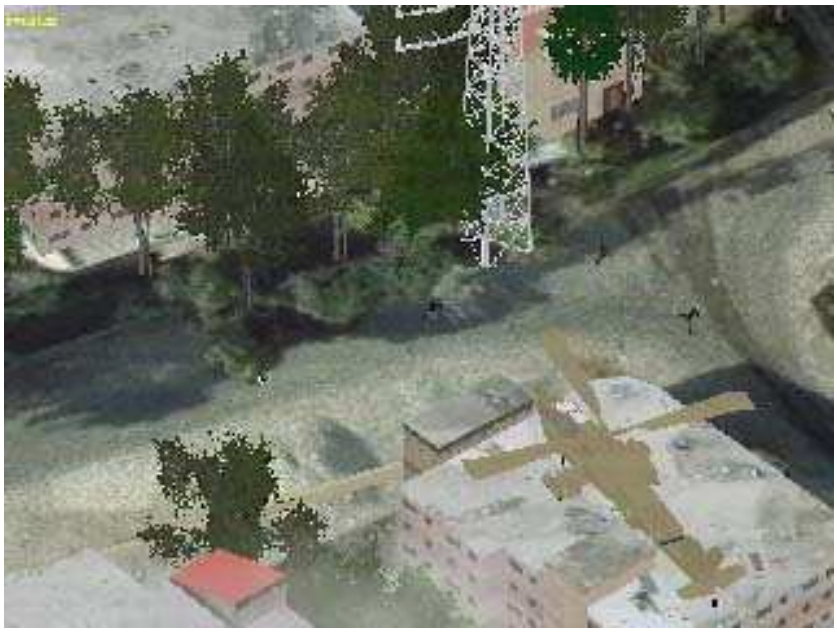


Figure 2.12: **Maintenance of team hierarchy: Response to events.**

	$Stop - execution$	$Elaborate - Next$	$No - Threat$	$Back - to - mission$
avrg	2.0055 (sec.)	4.018833333 (sec.)	12.06416667 (sec.)	2.014 (sec.)
stdv	0.021473239	0.000408248	0.002136976	0

Table 2.2: **Maintenance of team hierarchy: Response to threat, avrg and stdv over 10 runs with 6 agent teams.**

to changes in the environment. Figure 2.12 shows an Helicopter patrolling over such a generated team. The team responds to the event by splitting up and fleeting towards closely found hiding places. After the danger is removed, the team can regroup, however, this is not mandatory. Individual agents can decide to abort the mission. In such cases, Teamwork maintenance is used in order to update the Team hierarchy and continue executing the mission. Only the agents included in the new formed group will be updated from then on in regard to inner team communication, allocation and synchronization. The other agents, will continue executing the mission individually.

Table 2.2, summarizes the results gathered from 10 scenario runs during which an helicopter intercepted the team planned route, and later, moved on. The first

	$Teammate - loss$	$Team - hierarchy - update$
avrg	2.03 (sec.)	4.0202 (sec.)
stdv	0.032449961	0.043014

Table 2.3: **Maintenance of team hierarchy: Response to Team loss, avrg and stdv over 10 runs with 6 agent teams.**

column shows the time passing from the moment the helicopter was initially updated on any agent input-link to the time the last agent stopped executing the mission. The second column shows the time passing from the initial brake in the mission execution to the moment the whole team splits and moves to response. The third column shows the time passing from the moment the threat is removed and until the whole team regroups. The fourth column shows the time passing from the team regrouping to the moment the whole team resumes the mission execution. The time frames shown in Table 2.2 have been collected before several communication optimizations where implemented, today time can be substantially reduced. In this scenario team maintenance is used twice, once in order to decompose the team and once to regroup it. However, there is a significant difference. Without team maintenance, Leaving the team would require a time consuming process of agreements between team members, however, with an underlying team maintenance mechanism, this is already taken care of. However, While leaving the team requires no agreement, joining still does, since it uses the regular team-protocol and thus takes longer. From these results, we can conclude than, that straight forward team maintenance, can be used sometimes instead of a heavy resource team protocol. This is shown in Table 2.3.

Table 2.3, summarizes the results gathered from 10 scenario runs during which a teammate was shot down. The first column refers to the time passing from the moment the teammate death was initially updated on any agent input-link to the time the last agent stopped executing the mission. The second column refers to the time passing from the initial brake in the mission execution to the moment the whole team updated the team hierarchy. As expected, since leaving the team requires no agreement, the results are similar to those found in Table 2.2, on the corresponding event.

2.7 Conclusions and Future Work

This section 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 [28]; and BITE'M , 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.

Chapter 3

An Integrated Development Environment and Architecture for Soar-Based Agents

3.1 Summary

It is well known how challenging is the task of coding complex agents for virtual environments. This difficulty in developing and maintaining complex agents has been plaguing commercial applications of advanced agent technology in virtual environments. In this chapter, we discuss development of a commercial-grade integrated development environment (IDE) and agent architecture for simulation and training in a high-fidelity virtual environment. Specifically, we focus on two key areas of contribution. First, we discuss the addition of an explicit recipe mechanism to Soar, allowing reflection. Second, we discuss the development and usage of an IDE for building agents using our architecture; the approach we take is to tightly-couple the IDE to the architecture. The result is a *complete* development and deployment environment for agents situated in a complex dynamic virtual world.

3.2 Introduction

It is well known how challenging is the task of coding complex agents for virtual environments. This has been a topic for research in many papers including [2, 33, 13, 6]. This difficulty in developing and maintaining complex agents has made adoption of cognitive architectures difficult in commercial applications of virtual environments. Thus many companies work with different variations of state machines to generate behaviors [3].

In this chapteer, we discuss development of a commercial-grade development environment and agent architecture for simulation and training in a high-fidelity virtual environment. We discuss architectural support for coding of a complex plan execution by a team of agents, in Soar, and discuss the differences in our approach from previous approaches to using Soar in such tasks.

Specifically, we focus on two key areas of contribution. First, we discuss the addition of an explicit recipe mechanism to Soar, allowing reflection. This allows a programmer to build Soar operators (units of behavior) that are highly reusable, and can reason about their selection and de-selection. We show how this mechanism acts as a decision-kernel allowing multiple selection mechanisms (simulating human social choices, domain knowledge, etc.) to all co-exist on top of it. The recipe mechanism generates possible alternatives: The choice mechanisms assign preferences to these. Soar then decides.

Second, we discuss the development and usage of an integrated development environment (IDE) to build agents using our architecture. The approach we take is to tightly-couple the architecture to the development environment, so that bugs—which in Soar can be notoriously difficult to find [30]—can be ironed out as they are written.

We demonstrate these efforts in a *complete* development environment for Soar agents, situated in a complex dynamic virtual world, used for realistic simulation and training. We attempt to draw lessons learned, and highlight design choices which we feel were important from the perspective of an industrial project.

3.3 Background

Our work was done as part of Bar Ilan University's collaboration with Elbit Systems, Ltd. The goal is to create a *smart* synthetic entity—an agent—which performs in a variety of simulated scenarios. Agents should operate autonomously, behaving as realistically as possible. The agents will enhance Elbit's training and simulation products.

The environments in which the agents are to function are usually complex environments, containing up to entire cities, and including accurate placement of objects. The initial focus of the project is towards the development of individual entities, possibly working in small groups. Figure 3.1 shows an example screenshot from an application use-case.

Figure 3.1: **Urban terrain**

Both the architecture and IDE for the agents must be oriented towards the development of configurable entities, driven by capabilities, personality and complex plans. Such a view reinforces the need for a flexible architecture, able to cope

with many parameters and configurations of large plans (composed of recipes with 100 up to 1000 inner behavior nodes). The architecture must support several distinct cognitive mechanisms (emotions, focus of attention, memory, etc.) running in parallel and interacting, in each and every virtual modeled cognitive entity.

We briefly introduce here the various components of our architecture, and the rationale behind its design. The next sections will discuss the foci of the chapter in depth.

One main difference between commercial and academic frameworks for multi-agent systems, is in the use of hybrid architectures. While in most academic work it is sometimes possible—indeed, desired—to include all levels of control using a unified representation or mechanism, this is clearly not the case when it comes to large scale industrial applications. No single architecture or technology in this case is sufficient. Moreover, it is often critical to be able to interact with existing underlying components. This might come as a demand from the customer who ordered the project, or (sometimes) as a way to promote other technology available within the company.

With respect to academic work, this view goes back to past research on agent architectures, such as the ATLANTIS [7] architecture, which is based on the observation that there are different rates of activity in the environment, requiring different technologies. In our work, we were inspired as well by the vast research and conclusions drawn from the RoboCup simulation league [27] and from past simulation projects conducted in Soar such as the IFOR project [33, 13].

Indeed, our industrial partners have developed a hybrid architecture in which many components that have to do with cognitive or mental attitudes are actually outside of the main reasoning engine, built in Soar. The guiding philosophy in deciding whether something should be done in the Soar component has been to leave (as much as possible) any and all mathematical computations outside of Soar, including all path planning and motion control. For example, we rely on a controller in charge of moving an agent on a specified path. Such controller can be assigned the movement of teams of agents, and can use different movement configurations while trying to keep relations and angles fixed between it's members. In making this choice, the project is setting itself apart from other similar projects, in which Soar was used to control entities at a much more detailed level of control [33, 13, 27].

We focus in this chapter on the Soar decision-making component, and its associated IDE. Both of these, with the other components of the system, are hooked up to a VR-Forces [26] simulation environment, a high-fidelity simulator utilizing DIS. It is used for large scale projects ranging air, ground and naval training such as TACOP [37].

Given the task of providing an agent development framework, several architectures for this type of application might come to mind: JACK [10], SOAR [28], UMPRS [23], JAM [11], etc. Soar [28] is among the few that has commercial support, and yet is open-source, making it a clear favorable candidate for our project.

Soar uses globally-accessible working memory, and production rules to test and modify it. Efficient algorithms maintain the working memory in face of changes to specific propositions. Soar operates in several phases, one of which is a decision phase in which all relevant knowledge is brought to bear, through an XML layer, to make a selection of an operator (behavior) that will then carry out deliberate mental (and sometimes physical) actions.

A key novelty in Soar is that it automatically recognizes situations in which this decision-phases is stumped, either because no operator is available for selection (*state no-change impasse*), or because conflicting alternatives are proposed (*operator tie impasse*). When impasses are detected, a subgoal is automatically created to resolve it. Results of this decision process can be chunked for future reference, through Soar's integrated learning capabilities. Over the years, the impasse-mechanism was shown to be very general, in that domain-independent problem-solving strategies could be brought to bear for resolving impasses [28].

Being a mixture between a reactive and a deliberative system, it is usually very easy to program rules (productions) in Soar, so that a short sequence will be triggered upon certain conditions. However, building a complex scenario involving multiple agents becomes somewhat of an overwhelming task. Debugging just seems to never end [1].

Soar uses globally-accessible working memory. Each rule is composed by a left and right sides. Simplified, the left side of the rule is in charge of testing whether specific conditions hold in this working memory, while the right side is in charge making changes to the working memory. Thus each rule in the system can

[1] We note that similar motivations have lead in the past to contributions in other directions, e.g., teamwork [32].

read, write, and modify the working memory, triggering or disabling the proposal of other rules, including itself. This means that each Soar programmer must have complete knowledge of all the rules, taking all previous written code into account each time a new rule is added.

Another facet is that Soar does not differentiate between the change an operator makes, and the actual state of the agent, and ties them as one by coding conventions. Since Soar operates through states, this means that each operator by definition is tied to the state the agent is in. In other words, naive Soar programming requires all agent behaviors to be re-programmed each time a behavior is to be applied in a slightly different state than initially anticipated by the programmer.

One of the first architectural changes we aimed for was to overcome this relation between states and operators. By doing so, we could make use of generic types, templates, and other byproducts such as the utilization of reflection. These proved to be valuable programming tools.

3.4 Soaring Higher

The approach we take is to provide a higher level of programming, built on Soar foundations and taking advantage of the underlying framework. The most important component of this layer is *recipes*—behavior graphs—representing a template (skeletal) plan of execution of hierarchical behaviors [14, 32]. The behavior graph is an augmented connected graph tuple (B, S, V, b_0) , where B is a set of task-achieving behaviors (as vertices), S, V sets of directed edges between behaviors $(S \cap V = \emptyset)$, and $b_0 \in B$ a behavior in which execution begins.

Behaviors is defined as $b_i \in B$:

1. Constant parameters, with respect to the program execution scope (such as b_i timeout , probability etc..).

2. Dynamic parameters, with respect to b_i execution scope (such as the event that triggered b_i preconditions).

3. Maintenance conditions [19], with respect to b_i execution scope.

4. Teamwork conditions [19], with respect to b_i execution scope.

5. Preconditions which enable its selection (the robot can select between enabled behaviors).

6. Endconditions that determine when its execution must be stopped.

7. Application rules that determine what b_i should do upon execution.

In [14] S sequential edges specify temporal order of execution of behaviors. A sequential edge from b_1 to b_2 specifies that b_1 must executed before executing b_2. A path along sequential edges, i.e., a valid sequence of behaviors, is called an *execution chain*. V is a set of vertical *task-decomposition* edges, which allow a single higher-level behavior to be broken down into execution chains containing multiple lower-level behaviors. At any given moment, the agent executes a complete path root-to-leaf through the behavior graph. Sequential edges may form circles, but vertical edges cannot. Thus behaviors can be repeated by choice, but cannot be their own ancestors.

Even using this representation, we faced several abnormal situations. For example, if a leaf behavior has precondition equal to its ancestors endcondition it might never be proposed, or worst, constantly be terminated prematurely. Solving such a problem at an IDE level, contradicts the need for behavior encapsulation. Another problematic aspect of such an architecture is that during an execution chain no alternatives are being considered. Switching from one execution chain to the other (given that they both derive from the same parent behavior), needs ending the whole execution chain, a process which is both time consuming, and sometimes harms the overall reactiveness of the system. This problem emerges even when using the Soar architecture as provided. We will not deal with proposed solutions since they are out of this thesis's scope. However, one specific proposal involving a reactive recipe mechanism running on top of the regular one, can be viewed as a higher level selection mechanism, and thus is similar to other selection mechanism discussed later in detail.

The recipe mechanism is responsible for proposing operators for selection. Through reflection, it examines the current recipe data structure (graph), and proposes all operators that are currently selectable, based on their precondition and position within the recipe graph. It efficiently schedule the proposal and retraction of generic behaviors given certain conditions. These behaviors, are specified

inside generic subtrees of plans, which in turn are gathered in large abstract sets of plans. When a Soar agent is loaded, it assembles its recipe structure at run-time by recursively deepening, arranging and optimizing it.

Additional mechanisms are added to guide selection between the proposed operators. Examples to such mechanisms include probabilistic behavior selection, teamwork, social comparison theory [17], individual and collaborative condition maintenance [19], etc. For example, since the recipe enables reflection, one of the mechanisms monitors other agents' actions by the mirroring of the recipe onto another inner Soar state and using translation of sensory data. This allows modeling of another agent's decision processes based on observation—a form of plan recognition. Another mechanism is in charge of teamwork and keeps the team synchronized and roles allocated, by the use of communication [19].

With respect to the IDE, we made use of a new state-of-the-art facilities such as refactoring and testing of agent applications. Instead of building the IDE from scratch, as is commonly done, we chose to utilize an existing IDE, thus taking advantage of well-tested available technology. Our IDE is object-oriented, facilitating coding by the use of pre-made templates, re-usability of components such as plans and behaviors, instead of wizards and graphical means of programming.

In Soar, productions are proposed due to changes in WMEs (Soar working memory). In a behavioral context, this means that each behavior can be triggered by a change, both internal (inner state change) or external (sensory data), and that each behavior can affect the overall conduction of the system. During early phases of development we chose an approach similar to that found in [32], by providing a middle layer between Soar inputs and operators. However, as mentioned, Soar's productions can be triggered by internal events as well. Thus, we chose to broaden the common ground between behaviors by substituting the translation layer with an event-based mechanism. All our behaviors' preconditions and endconditions are triggered by explicit predicates, which signal events that are true. These events correspond to percepts, deduced or processed facts, and internal changes. They constitute explicit facts, internally classified by subject and category (e.g., all audio-related events groups together).

Adding events to Soar allows our agent means of reflection. A regular Soar agent is unaware of the actual change in the environment that lead to a specific operator instantiation, thus could not refer to the cause of it following a specific

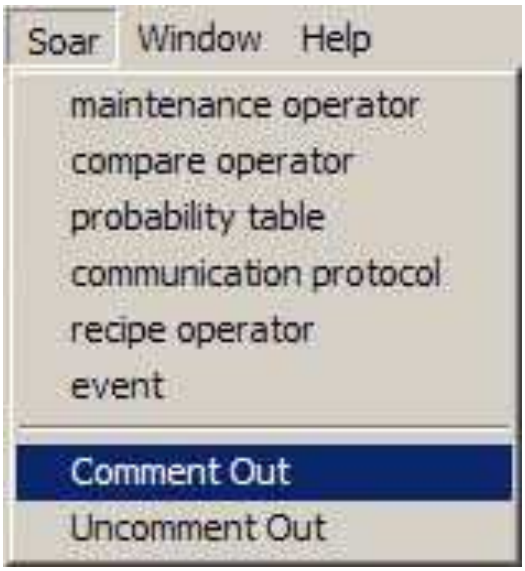


Figure 3.2: **Soar integrated templates**

sequence of actions. At most, it can reflect on the actions themselves. Using the event mechanism, however, allows the agent to consider exactly what changes lead to each operator/behavior proposal or termination, as the preconditions and endconditions are defined explicitly.

We found this approach critical when in need of communications between agents. The language by which our agents communicate is an event language: Entire subtrees of Soar working memory are being passed on and forth between agents. The agents thus pass between them sets of events relevant to the proposal or retraction of behaviors. This allows allocation of roles, and synchronization of the execution of behaviors. and

In our target environment, both recipe operators (task and maintenance) and events can be programmed with the help of code templates. During the coding phase we discovered that most bugs result from WME misspelling or errors in structure reference. Figure 3.2 shows the interface by which a user can automatically generate the appropriate operator or event code. Events are generated and categorized in different folders, classified by the inputs that trigger them or by the events that they relay on. Operators (behaviors) are generated with parameters, preconditions and endconditions, fully documented. This feature results in a clean uniform code, and thus simplifies debugging a great deal. Additional sup-

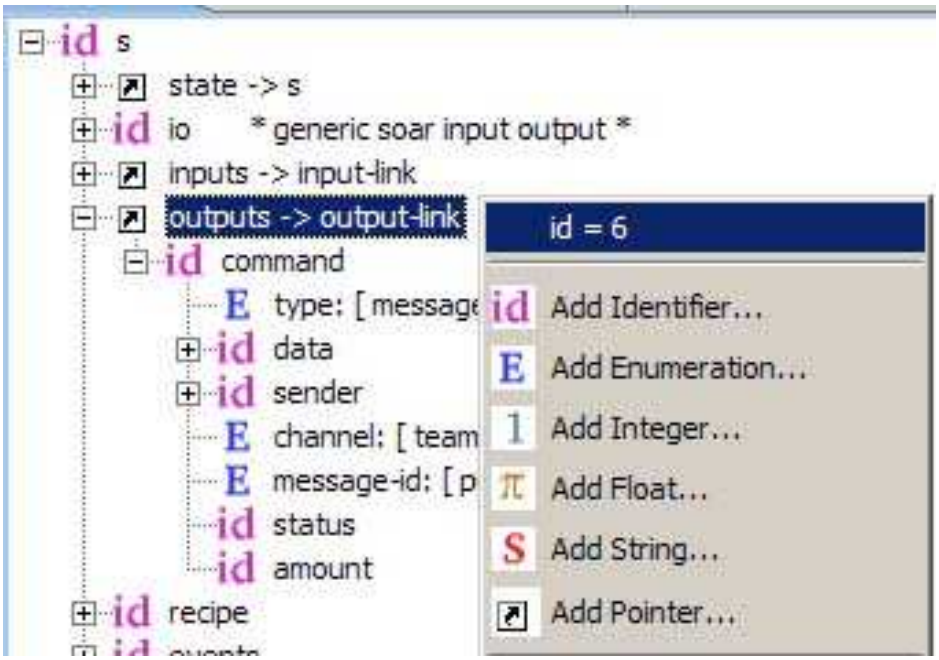


Figure 3.3: **Soar Datamap view**

port for communication protocols and probability tables for operator proposals is also provided.

The use of templates in Soar, goes back to the early IDE development tools for Soar agents. Our tool differs from those earlier works in that it provides not only basic support for Soar operator application and proposal rules templates, but an extensive elaborated behavior structure supported by the recipe mechanism, specialized for the architecture we use. The use of the templates saves much coding for the programmer, since they already encapsulate much of what the programmer needs to consider.

The Soar datamap is a representation of the Soar memory structure generated through the execution of a Soar program, and can be inferred by the left side and right side of Soar production rules. Several tools are available in order to generate a Soar datamaps through static analysis of Soar productions. We significantly extended the initial Eclipse extension for datamap support, provided by the University of Michigan and SoarTech, adding additional services and tools. Most of our coding tools now rely on the datamap, enabling us to generate specific insight-

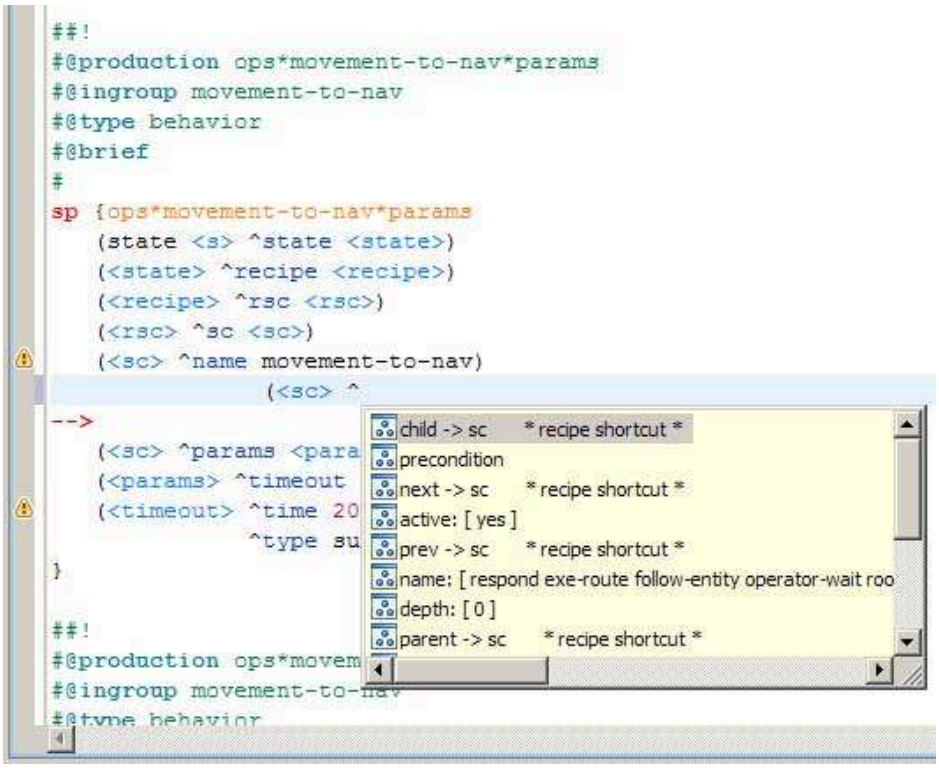


Figure 3.4: **Auto complete with deep inspection**

ful warnings, provide smart assistance, and auto completion of code that takes the structure of the memory in our architecture into account.

Aside from warnings and code assistance, Soar benefits from the many Eclipse plug-ins that are already present and developed within the IDE environment. Among those are integrated documentation support, execution of Soar agents, and integrated debugger. Support for both VSS and CVS code-versioning systems can be found as well, for large team projects.

We have also extended and enhanced the SoarJavaDebugger which is distributed with the current version of Soar, by the University of Michigan. The first customization, seen in Figure 3.5, utilizes a UML type of visualization, in order to display the recipe at run time. At any point the active path to the currently executed behavior is presented along with optional behaviors not chosen (colored red). These red behaviors have matching preconditions, but were not activated due to hierarchical or situational constraints. This recipe visualization is updated as well at runtime, enabling the programmer to focus only on the relevant executed

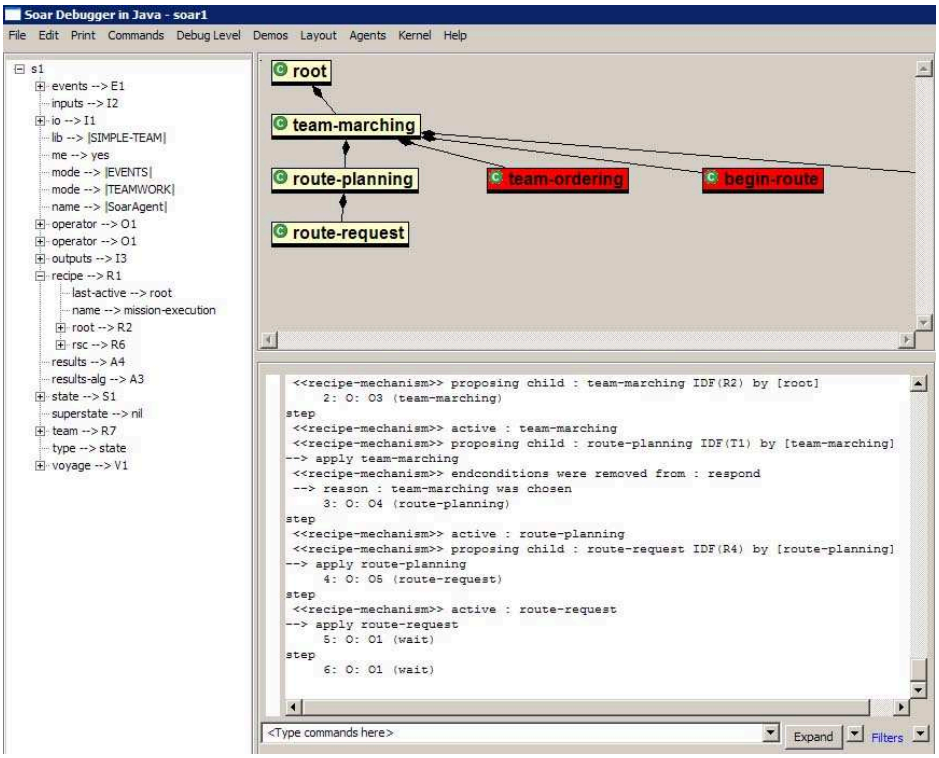


Figure 3.5: **Soar Java Debugger, with additional Tree View and Recipe Visualization**

subset of the recipe.

In addition, on the left-hand side of the debugger window, is a tree-folder view of the working memory. The root of the tree can be set to point any subset of the agent knowledge (any Working Memory Element, WMEs) and is updated at runtime. Since Soar already arranges WMEs in a tree like format, it greatly speeds up debugging to be able to inspect the agent knowledge by simply clicking such folders.

3.5 Evaluation

Evaluation of the contributions described above is challenging. The first contribution involves the use of reflection in the recipe, which allows clean separation of the process by which the knowledge of the agent proposes alternatives, and the mechanisms that facilitate a decision among them. During the evaluation of our system, we made use of a scenario in which a team of agents uses communications to agree upon several mission points. They calculate routes considering possible threats along the way and travel from one location to the other. While doing so they collaboratively maintain several movement protocols and react to changes in the environment such as the appearance of new threats, the loss of team members, etc.. During the execution of the scenario, the agents move from one waypoint to another, maintaining specified formations, and reorganize these formations given changes in the team hierarchy.

To provide some insight as to the performance of the design, we compare our system to previous systems that have utilized Soar as their basis. The most well-known similar system is TacAir-Soar, a highly successful project using Soar as the basis for synthetic pilots, capable of running a wide variety of missions [33, 13]. Less complex—yet still successful—applications of Soar included the ISIS-97 and ISIS-98 RoboCup teams [34].

Table 3.1 provides a comparison of key features, allowing a qualitative insight into the complexity of these systems, compared to the system discussed in this chapter. The columns report (left-to-right) on the overall number of Soar rules used in the system, the number of unique actions (outputs), the amount of unique percepts (inputs), and the number of actual domain/task behaviors/operators.

Our system, at its current state of development, is of moderate complexity

architecture	*rules*	*actions*	*inputs*	*operators*
TacAir-Soar	5200	30	200	450
ISIS-97/98	1000	7	50	40
Ours	650	25	200	100

Table 3.1: **Architectural Complexity Evaluation**

compared to efforts that have been reported in the literature. Taking the combined inputs and outputs as the a basic measure of the complexity of the task, would put our system's task on par with that of the TacAir-Soar system, and far ahead of the challenge faced by RoboCup teams. However, looking at the number of operators, we see that the knowledge of our agents, while still significantly more complex than that of the RoboCup agents, is still very much behind that of the advanced TacAir Soar.

Based on this qualitative assessment, which puts our system somewhere in the middle between the TacAir-Soar and the ISIS systems, it is interesting to note that our system uses significantly less rules than *both* other systems, to encode the knowledge of the agents. While we use about 6.5 rules, on average, for supporting each operator, TacAir-Soar uses 11.5 and RoboCup about 25. We believe that this is due, at least in part, to the use of the recipe mechanism. In both previous systems, the preconditions of operators tested not only the appropriateness of an operator given the mental attitude of the agents with respect to its environment and goals, but also with respect to the position of the operator compared to other task operators. For instance, commonly operators would have to test for the activation of their parents, before being proposed. The recipe mechanism cleanly separates the two.

On our system, operator rules only determine whether the task-related preconditions of an operator have been satisfied. The rules proposing the operator if its preconditions are true, *and given its position within the behavior graph*, are all part of the recipe mechanism. We believe that this saves a significant number of rules.

It also saves significant programming effort: Since our operators do not refer at any point to their execution point, changing the occurrence of a generic action (or a generic subtree of hierarchical actions within a recipe) requires only updating the configuration of the Soar coded recipe. In comparison, moving operators around in previous systems, from one specific execution point to another

(one point in the recipe to the other) would require changes to be made in all branching children (all rules testing the occurrence of such an operator), since the hierarchy is part of each sub-operator's preconditions. Additionally, by previous Soar conventions, operator source files were written in hierarchical file system, which reflected the intended hierarchical decompositions. Moving operators in the recipe either caused files to move around, or worse yet, created a discrepancy between the convention of the file-system and the position of the operator in memory. Freeing Soar operators from their execution point also allowed us to place all operator files in a single directory, making finding and maintaining them much easier.

Previous Soar architectures, have utilized a specific style of writing in Soar, in which hierarchical decompositions are created in memory by relying on Soar's *operator no-change* impasse to keep track of the active hierarchical decomposition. But the creation and maintenance of impasses can be expensive. The recipe mechanism allows us efficient book-keeping of the current decomposition, without using impasses (unless needed for other reasons).

To demonstrate the savings offered by using the recipe mechanism, Table 3.2 provides data gathered from the execution of several standard Soar benchmarks (bundled with the Soar architecture distribution), on the same hardware and software configuration (Soar 8.6.2 kernel on the same Pentium 4 CPU 3.2 GHz 512MB ram). These standard problems consisted of the Missionaries and Cannibals (MaC) problem, and the performance of 1000 random arithmetic calculations. We compare Soar's performance in both, with the test scenario, described above.

Table 3.2 consists of four columns: The number of decision-cycles in Soar (input to output phase) using an average run, the average time for each decision-cycle in milliseconds, the average size of Soar working memory at any time, and the number of changes to this memory. As shown, our architecture is much faster than the benchmarks—despite their simplicity relative to the task our system faces. Our decision-cycles are substantially faster mainly due to the recipe mechanism (which avoids impasses) and the utilization of controllers. Though new input is constantly delivered to our agents, most of the time our agent is idle, waiting for the current operator/behavior execution, the proposal of new behaviors or the arrival of critical data. Such results are crucial for demonstrating the scalability of

$benchmark$	dc	$msec/dc$	$WM(mean, changes)$
MaC	200	0.155	(49.896,13651)
Arithmetic	41487	0.320	(983.589,879076)
Ours	31363	0.078	(3266.797,196263)

Table 3.2: **Runtime Evaluation**

the system, for future scenarios (e.g., those simulating crowds).

We now turn to evaluation of the integrated development environments. As one could expect, quantitative evaluation is difficult here. Not only is the impact of the changes difficult to measure directly, but the target audience—Soar programmers—is very small. Nevertheless, we asked our current users to provide qualitative feedback on the tool, and compare it to previously-published development tools for Soar (such as Visual Soar, which is packaged with the Soar distribution).

Our users varied in experience, and in responses. One veteran Soar programmer has previously developed in Soar using emacs text-editor (without any GUI support for debugging), and later in Visual Soar. His assessment was that the use of the Eclipse environment was a marked improvement over Visual Soar (which, not surprisingly, was believed to be a significant improvement over emacs). He reported that the use of templates was not a speed-saver: As a veteran Soar programmer, he was used to writing code directly, without templates. On the other hand, two relatively novice programmers now swear by the Eclipse environment, and show strong preference to it over existing tools. They report that the templates are very useful, though they lose some of the usefulness over time. Based on these qualitative reports, it is clear that in an industrial project, a strong IDE such as Eclipse, is a valuable tool which provides many benefits in comparison to the alternatives.

3.6 Conclusion

In this chapter we discussed both the architecture and development environment for computed generated forces, based on an extended version of Soar. On an architectural level we proposed the addition of an explicit recipe mechanism to Soar, allowing reflection. This allows a programmer to build Soar operators (units of behavior) that are highly reusable and effective. We proposed how such a mech-

anism could act as a decision-making kernel. Second, we discussed the development and usage of an integrated development environment (IDE) to build agents using our architecture. We attempted to draw lessons learned, and highlight design choices which we felt were important from the perspective of an industrial project. We believe those insights can contribute towards the future development of computer-generated forces, in complex dynamic virtual worlds.

Bibliography

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

[2] R. Bordini, L. Braubach, M. Dastani, A. E. F. Seghrouchni, J. Gomez-Sanz, J. Leite, G. O'Hare, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems. In *Informatica 30*, pages 33–44, 2006.

[3] R. B. Calder, J. E. Smith, A. J. Courtemanche, J. M. F. Mar, and A. Z. Ceranowicz. Modsaf behavior simulation and control. In *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Reresentation*, Orlando, Florida, March 1993. Institute for Simulation and Training, University of Central Florida.

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

[5] S. Duff, J. Harland, and J. Thangarajah. On proactivity and maintenance goals. In *AAMAS '06: Proceedings of the Fifth international joint conference on Autonomous agents and multiagent systems*, pages 1033–1040, New York, NY, USA, 2006. ACM.

[6] T. D.Vu, J. Go, G. A. Kaminka, M. M. Veloso, and B. Browning. MONAD: A flexible architecture for multi-agent control. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-03)*, 2003.

[7] E. Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the*

Tenth National Conference on Artificial Intelligence (AAAI-92). Menlo Park, Calif.: AAAI press, 1992.

[8] 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.

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

[10] N. Howden, R. Rönnquist, A. Hodgson, and A. Lucas. JACK: Summary of an agent infrastructure. In *Proceedings of the Agents-2001 workshop on Infrastructure for Scalable Multi-Agent Systems*, 2001.

[11] M. J. Huber. JAM: A BDI–theoretic mobile agent architecture. In *Proceedings of the Third International Conference on Autonomous Agents (Agents-99)*, pages 236–243, 1999.

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

[13] R. M. Jones, J. E. Laird, N. P. E., K. Coulter, P. Kenny, and F. Koss. Automated intelligent pilots for combat flight simulation. *AI Magazine*, 20(1):27–42, 1999.

[14] G. A. Kaminka and I. Frenkel. Flexible teamwork in behavior-based robots. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, 2005.

[15] G. A. Kaminka and I. Frenkel. Integration of coordination mechanisms in the BITE multi-robot architecture. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA-07)*, 2007.

[16] G. A. Kaminka and N. Fridman. Social comparison in crowds: A short report. In *Proceedings of the Sixth International Joint Conferance on Autonomous Agents and Multi-Agent System (AAMAS-07)*, 2007.

[17] G. A. Kaminka and N. Fridman. Social comparison in crowds: A short report. In *Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-07)*, 2007.

[18] 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.

[19] G. A. Kaminka, A. Yakir, D. Erusalimchik, and N. Cohen-Nov. Towards collaborative task and team maintenance. In *Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-07)*, 2007.

[20] G. A. Kaminka, A. Yakir, D. Erusalimchik, and N. C. Nov. Towards collaborative task and team maintenance. In *Proceedings of the Sixth International Joint Conferance on Autonomous Agent and Multi-Agent System (AAMAS-07)*, 2007.

[21] S. Kumar and P. R. Cohen. Towards a fault-tolerant multi-agent system architecture. In *Proceedings of the Fourth International Conference on Autonomous Agents (Agents-00)*, pages 459–466, Barcelona, Spain, 2000. ACM Press.

[22] S. Kumar, P. R. Cohen, and H. J. Levesque. The adaptive agent architecture: Achieving fault-tolerance using persistent broker teams. In *Proceedings of the Fourth International Conference on Multiagent Systems (ICMAS-00)*, pages 159–166, Boston, MA, 2000. IEEE Computer Society.

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

[24] 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.

[25] N. Lesh, C. Rich, and C. L. Sidner. Using plan recognition in human-computer collaboration. In *Proceedings of the Seventh International Conference on User Modelling (UM-99)*, Banff, Canada, 1999.

[26] MÄK Technologies. VR-Forces. http://www.mak.com/vrforces.htm, 2006.

[27] S. C. Marsella, J. Adibi, Y. Al-Onaizan, G. A. Kaminka, I. Muslea, M. Tallis, and M. Tambe. On being a teammate: Experiences acquired in the design of robocup teams. In *Proceedings of the Third International Conference on Autonomous Agents (Agents-99)*, pages 221–227, Seattle, WA, 1999. ACM Press.

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

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

[30] Ritter, F. E., G. P. Morgan, Stevenson, W. E., and M. A. Cohen. A tutorial on Herbal: A high-level language and development environment based on protï¿½ï¿½for developing cognitive models in soar. In *Proceedings of the 14th Conference on Behavior Representation in Modeling and Simulation*. 2005.

[31] Soar homepage. http://sitemaker.umich.edu/soar/home/, 2006.

[32] M. Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, 7:83–124, 1997.

[33] M. Tambe, W. L. Johnson, R. Jones, F. Koss, J. E. Laird, P. S. Rosenbloom, and K. Schwamb. Intelligent agents for interactive simulation environments. *AI Magazine*, 16(1), Spring 1995.

[34] M. Tambe, G. A. Kaminka, S. C. Marsella, I. Muslea, and T. Raines. Two fielded teams and two experts: A robocup challenge response from the trenches. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-99)*, volume 1, pages 276–281, August 1999.

[35] M. Tambe, D. V. Pynadath, N. Chauvat, A. Das, and G. A. Kaminka. Adaptive agent integration architectures for heterogeneous team members. In *Proceedings of the Fourth International Conference on Multiagent Systems (ICMAS-00)*, pages 301–308, Boston, MA, 2000.

[36] M. Tambe and W. Zhang. Towards flexible teamwork in peristent teams. In *Proceedings of the Third International Conference on Multiagent Systems (ICMAS-98)*, 1998.

[37] W. A. van Doesburg, A. Heuvelink, and E. L. van den Broek. Tacop: a cognitive agent for a naval training simulation environment. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-05)*, pages 34–41, New York, NY, USA, 2005. ACM Press.

[38] A. Yakir, G. A. Kaminka, D. Erusalimchik, and N. C. Nov. An integrated development environment and architecture for soar-based agents. In *Innovative Applications of Artificial Intelligence (IAAI-07)*, 2007.

[39] J. Yen, J. Yin, T. R. Ioerger, M. S. Miller, D. Xu, and R. A. Volz. CAST: Collaborative agents for simulating teamwork. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 1135–1144, 2001.