
Autonomous Agents and Multi-Agent Systems, 7, 71–100, 2003

An Automated Teamwork Infrastructure for Heterogeneous Software Agents and Humans

DAVID V. PYNADATH pynadath@isi.edu

Information Sciences Institute and Computer Science Department, University of Southern California, 4676 Admiralty Way, Marina del Rey, CA 90292

MILIND TAMBE tambe@isi.edu

Information Sciences Institute and Computer Science Department, University of Southern California, 4676 Admiralty Way, Marina del Rey, CA 90292

Abstract. Agent integration architectures enable a heterogeneous, distributed set of agents to work together to address problems of greater complexity than those addressed by the individual agents themselves. Unfortunately, integrating software agents and humans to perform real-world tasks in a large-scale system remains difficult, especially due to three main challenges: ensuring robust execution in the face of a dynamic environment, providing abstract task specifications without all the low-level coordination details, and finding appropriate agents for inclusion in the overall system. To address these challenges, our Teamcore project provides the integration architecture with general-purpose teamwork coordination capabilities. We make each agent *team-ready* by providing it with a proxy capable of general teamwork reasoning. Thus, a key novelty and strength of our framework is that powerful teamwork capabilities are built into its foundations by providing the proxies themselves with a teamwork model.

Given this teamwork model, the Teamcore proxies addresses the first agent integration challenge, robust execution, by automatically generating the required coordination actions for the agents they represent. We can also exploit the proxies' reusable general teamwork knowledge to address the second agent integration challenge. Through *team-oriented programming*, a developer specifies a hierarchical organization and its goals and plans, abstracting away from coordination details. Finally, KARMA, our Knowledgeable Agent Resources Manager Assistant, can aid the developer in conquering the third agent integration challenge by locating agents that match the specified organization's requirements. Our integration architecture enables teamwork among agents with no coordination capabilities, and it establishes and automates consistent teamwork among agents with some coordination capabilities. Thus, team-oriented programming provides a level of abstraction that can be used on top of previous approaches to agent-oriented programming. We illustrate how the Teamcore architecture successfully addressed the challenges of agent integration in two application domains: simulated rehearsal of a military evacuation mission and facilitation of human collaboration.

Keywords: multiagent systems, teamwork, agent integration architectures.

1. Introduction

An increasing number of agent-based systems now operate in complex dynamic environments, such as disaster rescue missions, monitoring/surveillance tasks, enterprise integration, and education/training environments. With this increasing population of available agents, we can expect another powerful trend: the reuse of specialized agents as standardized building blocks for large-scale systems [11,13,16]. This prediction is based on two observations. First, developers continue to construct software systems out of ever-larger reusable components, rather than as monoliths [31]. The reuse of agents as components is

the next logical step — enabling a richer reuse mechanism than component-ware or application frameworks [16]. Second, there is the already growing trend of integration of agent components in cooperative information systems, networked embedded systems, and other software systems [12, 13, 30].

System designers can integrate these existing agents to construct new multi-agent systems capable of solving problems of greater complexity than those addressed by the individual agents themselves. Agent integration architectures enable a heterogeneous, distributed set of agents to work together to address such large-scale problems. Such integration architectures face an ever-increasing variety of available agents. In addition, as these agents move into more and more domains that require human interaction, these integration architectures must also tackle the coordination of people who exhibit a diversity and complexity beyond that of even software agents.

Unfortunately, integrating agents to perform real-world tasks in a large-scale system remains difficult. There are at least three key challenges that agent integration architectures face. First, it is difficult to ensure robust and flexible execution of the desired tasks. In addition to the risk of failures among individual agents, an integrated system also runs the risk of coordination breakdowns, due to (for instance) one agent's lacking key information known to the others. However, in an open environment, we cannot expect agents to come ready-made to avoid such breakdowns. Furthermore, even if an agent is capable of proper coordination (as is the case with people), it is often preferable for the architecture to take on some of this burden and free the agent or person to direct its resources to its individual tasks. Second, in general, building an integrated system to accomplish robust execution can require a potentially large number of coordination plans to cover all of the low-level coordination details. This problem is further exacerbated when the designer must create new plans for each new systemwide task or set of agents. Third, it is often difficult to locate and recruit relevant agents for integration in a distributed, open environment. There are other challenges as well (e.g., agent communication languages), but this article focuses on the three listed here.

We begin with a discussion of the coordination challenge in agent integration. To address this challenge, our architecture, called Teamcore,[1] focuses on general-purpose teamwork capabilities. Existing theories of teamwork, such as joint intentions [6] and SharedPlans [9], provide an analytical framework for designing coordination behavior with strong guarantees. This research laid the theoretical groundwork for implemented systems that demonstrated the real-world utility of teamwork in designing robust organizations of agents that coordinate amongst themselves [14, 32, 33]. Based on these successful applications of teamwork to closed multiagent systems, the key hypothesis behind Teamcore is that teamwork among agents can enhance robust execution even among heterogeneous agents in an open environment. No matter how diverse the agents may be, if they act as team members, then we can expect them to act responsibly towards each other, to cover for each other's execution failures, and to exchange key information.

Therefore, our integration architecture is founded on powerful in-built teamwork capabilities. Essentially, the architecture enables teamwork among agents with no coordination capabilities, and it establishes and automates consistent teamwork among agents with some coordination capabilities, by providing them with a proxy capable of general teamwork reasoning. Each proxy contains a general teamwork model for such reasoning, which it uses to provide consistent *team readiness* to the heterogeneous agent it

represents. Since team members behave responsibly towards each other, a team formed using such proxies can achieve its goals robustly, with agents automatically covering for failed teammates, supplying key information to help each other, etc. The novelty of our architecture stems from these in-built teamwork capabilities that provide the required robustness and flexibility in agent integration, without requiring modification of the agents themselves. This contrasts with other architectures such as OAA [20] that provide centralized facilitators, but require the hand-generated addition of such teamwork capabilities to the agents being integrated. The distributed nature of Teamcore also avoids any centralized bottlenecks and central points of failure.

We have implemented our Teamcore architecture using STEAM [32] as the proxies' teamwork model. STEAM provides a reusable, general-purpose teamwork module that encapsulates reasoning about common teamwork coordination, including contingencies in such coordination. STEAM has already proven effective in multiple coordination domains, so it forms a natural basis for providing normative teamwork capabilities in the more open, heterogeneous environments that Teamcore addresses. Given the STEAM module, the Teamcore proxies automatically generate the required coordination actions in executing their tasks. They communicate amongst themselves to ensure coherent execution of the tasks and to disseminate relevant information to the appropriate team members.

Beyond the forms of failures covered by their teamwork model, the Teamcore proxies must also handle other threatening contingencies (e.g., small changes in agents' response times during a run). To address such contingencies, the Teamcore proxies adapt to the needs and performance of specific individual agents. Teamcore's teamwork knowledge provides a critical foundation for this adaptation, by providing a layer of abstraction that structures the learning process — learning coordination knowledge from scratch for each and every possible contingency is very expensive in general. Currently, the proxies adapt in two different ways to the agents they represent. First, agents vary in the degree to which they can perform their own coordination actions. Each Teamcore proxy must have *adjustable autonomy* and adapt its level of decision-making autonomy in its team activities, so that it defers the appropriate decisions to the human or agent it represents. Second, during runtime in a dynamic environment, an agent's performance may deviate from its normal behavior (e.g., an agent's response time suddenly grows longer). For robust execution of the overall task, the proxies can perform *dynamic plan alteration*, to adapt to the current performance of the agents involved.

Because the Teamcore proxies automatically generate the required coordination actions in executing their plans, they shield the human developer from the second agent integration challenge of generating all of the low-level coordination details by hand. With the Teamcore architecture, we can exploit the proxies' reusable general teamwork knowledge to support abstract plan specification through *team-oriented programming*. Through team-oriented programming, a developer specifies a hierarchical organization and its goals and plans, abstracting away from coordination details. KARMA, our Knowledgeable Agent Resources Manager Assistant, can aid the developer in conquering the third agent integration challenge by locating agents that match the specified organization's requirements and assisting in allocating organizational roles to these agents. Team-oriented programming provides a level of abstraction that can be used on top of previous approaches to agent-oriented programming [28].

Section 2 motivates the requirements for agent integration by describing the two multiagent domains to which we have applied the Teamcore architecture. Section 3 describes the coordination capabilities of the Teamcore proxies and how they enforce robust execution. Section 4 describes how a software developer can design a multiagent organization through team-oriented programming. Section 5 describes Teamcore's various degrees of adaptivity. Section 6 evaluates our framework's ability to address the integration challenges presented by the two domains. Section 7 compares other agent integration architectures related to Teamcore. Section 8 summarizes the contributions of the work presented here.

2. Motivation

This paper describes, illustrates, and discusses the Teamcore framework using two concrete examples — evacuation of civilians stranded in a hostile area and human collaboration — where we have successfully applied this framework. The rest of this section provides more detailed descriptions of these two domains.

2.1. Application 1: Evacuation rehearsal

In the evacuation domain, the goal is an integrated system for simulated mission rehearsal of the evacuation of civilians from a threatened location. The system must enable a human commander to interactively provide locations of the stranded civilians, safe areas for evacuation, and other key points. A set of simulated helicopters should fly a coordinated mission to evacuate the civilians. The integrated system must plan routes to avoid known obstacles, dynamically obtain information about enemy threats, and change routes when needed. The following agents were available:

- **Quickset:** (from P. Cohen et al., Oregon Graduate Institute) Multimodal command input agents [C++, Windows NT] [5]
- **Route planner:** (from Sycara et al., Carnegie-Mellon University) Retsina path planner for aircraft [C++, Windows NT] [30]
- **Ariadne:** (from Minton et al., USC Information Sciences Institute) Database engine for dynamic threats [Lisp, Unix] [18]
- **Helicopter pilots:** (from Tambe, USC Information Sciences Institute) Pilot agents for simulated helicopters [Soar, Unix]

As this list illustrates, the agents are developed by different research groups, they are written in different languages, they run on different operating systems, they may be distributed geographically (e.g., on machines at different universities), and they have *no pre-existing teamwork capabilities*. There are actually 11 agents overall, including the Ariadne, route-planner, Quickset, and eight different helicopters (some for transport, some for escort). These agents provided a fixed specification of possible communication and task capabilities. Thus, the challenge in this domain lies in getting this diverse set of distributed agents to work together, without directly modifying the agents themselves.

2.2. Application 2: Assisting human collaboration

We have also applied Teamcore to assist human collaboration in our research team by automating many of our routine coordination tasks. Here, the agents to be integrated are members of our research group. The proxies know their users' scheduled meetings (by monitoring their calendars) and their whereabouts (e.g., whether they are working at their workstations). The Teamcore proxies must then assist in robust execution of team activities such as meetings. For example, if a user is still working at his/her workstation at the time of the meeting (e.g., to finish a paper), others should be automatically informed of an appropriate meeting delay. The overall system must also assign people to roles within team activities (e.g., selecting someone to give a presentation at a weekly research group meeting).

This system faces the daunting challenges of the users' heterogeneity (e.g., different presentation capabilities for different topics) and the larger scale of the team activities (e.g., each person is a member of multiple subgroups and has multiple meetings). In addition, the system cannot simply assign tasks for people, as it would the software agents of the evacuation domain. The system must also provide reliable communication with the users to perform these coordination tasks. One interaction mechanism available is the use of dialog boxes on the user's workstation display. Within our research group, five members currently have PDAs or WAP-enabled cellular phones that the system can also exploit for interactions. In addition, the PDA can also provide location information if connected to a Global Positioning System (GPS) device (as in Figure 1). As a final means of communication, a proxy can send email to a project assistant or some other third party who can contact the user directly to pass on the message.

Since the initial submission of this article, this domain has evolved into the "Electric Elves" domain [4, 27] and has been reported in the popular press as well (USA Today, http://www.usatoday.com).

3. Teamcore: Robust execution of team plans

Figure 2 shows the overall Teamcore framework for building agent organizations. The numbered arrows show the typical stages of interactions in this system. In stage 1, human

Figure 1. PDA (Palm VII) with GPS device and WAP-enabled cellular phone for wireless, handheld communication between proxy and user.

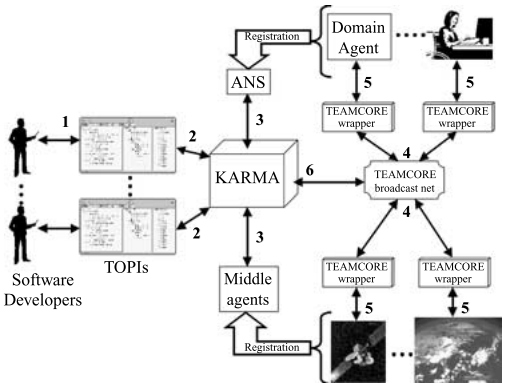

Figure 2. Teamcore framework: Teamcore proxies for heterogeneous domain agents.

developers interact with a team-oriented programming interface (TOPI) to specify a team-oriented program, consisting of an organization and its team plans. TOPI communicates this specification to KARMA in stage 2. In stage 3, KARMA derives the requirements for roles in the organization, and searches for agents with relevant expertise (called *domain agents* in Figure 2). To this end, KARMA queries different middle agents, white pages (Agent Naming Service), etc. Once it has located these domain agents, KARMA further assists a developer in assigning agents to organizational roles.

Having thus fully defined a team-oriented program, the developer launches the Teamcore proxies that jointly execute the team plans of the team-oriented program. To perform the coordination necessary for this execution, the proxies broadcast information among themselves via multiple broadcast nets (stage 4). The Teamcore proxies execute the team plans and, in the process, also generate specific requests and process the replies of their domain agents (stage 5). KARMA also eavesdrops on the various broadcasts to monitor the Teamcore proxies' progress (stage 6), which it displays to the software developer for debugging purposes. All communication among Teamcore proxies, between a domain agent and its Teamcore proxy, and between a Teamcore proxy and KARMA currently occurs via the KQML agent communication language [8].

To ensure robust execution, the Teamcore architecture transforms agents of all types into a set of consistent team players. As described in Section 1 and as illustrated in Figure 2, we achieve this team readiness among heterogeneous agents by providing each agent with a Teamcore proxy. The distributed Teamcore proxies, based on the Soar [21] rule-based integrated agent architecture, execute their joint plans in a distributed fashion and coordinate as a team during this execution. Section 3.1 describes the contents of the STEAM module that encodes the definition of team readiness that the Teamcore proxies use in coordination. Section 3.2 describes how the Teamcore proxies apply this definition in coordinating the actions of their domain agents.

3.1. STEAM: Making heterogeneous agents team ready

Each proxy contains the STEAM domain-independent teamwork module, responsible for Teamcore's teamwork reasoning [32, 33]. The STEAM algorithm is specified in detail in

Appendix A. We have implemented the algorithm using production rules in Soar [21], with examples shown in Appendix B. Implementations of the algorithm in other architectures are also possible. We can categorize the rules as providing three different types of high-level functionality, as discussed below:

Coherence preserving rules require team members to communicate with each other to ensure coherent initiation and termination of team plans. Coherent initiation ensures that all members of the team begin joint execution of the same team plan at the same time. Therefore, these rules prevent a helicopter from flying to its destination before all the other members of its flight team are ready to begin as well. Coherent termination requires that a team member inform others if it uncovers crucial information. We define "crucial information" as any condition that indicates that the team plan is achieved, unachievable, or irrelevant. For instance, the rules prescribe that anyone who is going to be late for a meeting must notify the other attendees, since the achievability of the meeting is now threatened.

Monitor and repair rules ensure that team members make an effort to observe the performance of their teammates, in case any of them should fail. If a critical team member (or subteam) should fail, we ensure that a capable team member (or subteam) takes over the role of the failed agent. For instance, the presenter at a research group meeting has a critical role with respect to the corresponding team plan, since without a presenter, the meeting will fail. Therefore, these rules specify that the team continuously monitor the presenter's ability to fulfill this role. If the presenter is unable to attend, these rules require that the team find some other capable team member to step in and give the presentation instead.

Selectivity-in-communication rules use decision theory to weigh communication costs and benefits to avoid excessive communication in the team. We thus ensure that the team performs coordination actions whose value in achieving coherent behavior outweighs the cost of communication. For instance, in the evacuation domain, communication is moderately expensive, due to the risk of enemy eavesdropping. Therefore, these rules would prescribe communication only when there is a sufficiently high likelihood and cost of miscoordination (e.g., transport helicopters arriving at rendezvous point at a different time from their escorts). Communication is much less costly in the human collaboration domain; however, the likelihood of miscoordination is also much lower, since the human team members perform some coordination actions themselves. For instance, the rules would *not* require communication to initiate a meeting plan, since all of the attendees have already entered the meeting into their calendar programs. On the other hand, the rules do require communication if an attendee is unable to arrive on time, since the other attendees are unlikely to know this information without any communication.

STEAM's 300 Soar rules are available in the public domain and have proven successful in several different domains reported in the literature. The novelty in the current work lies in the extensions to STEAM that enable the application of its rules to a much broader class of agents and problem domains.

3.2. Teamcore's interface with domain agents

In previous work [32, 33], STEAM resided directly in the domain agent's knowledge base, which is often difficult (if not impossible) to implement in an open, heterogeneous

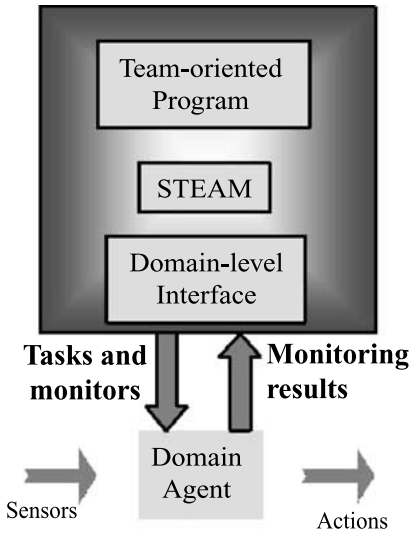

Figure 3. Reasoning components of a Teamcore proxy and interactions with domain agent.

environment. By placing STEAM's teamwork knowledge (rules) in a separate Teamcore proxy, we no longer need to modify code in the domain agent. However, the Teamcore proxy must now contain an interface module for communication with the domain agent, as illustrated in Figure 3.[2] In particular, the STEAM rules enable the Teamcore proxies to automatically communicate with each other to maintain team coherence and recover from member failures. In contrast, the interface module enables a Teamcore proxy to communicate with its domain agent, by translating the state of the team's execution into individual tasks and monitoring requests.

The Teamcore proxy generates task requests according to the role assigned to its corresponding domain agent in the organization. If the overall state of the team's execution requires that a domain agent now perform a particular task, its Teamcore proxy generates the appropriate request message, based on the domain agent's interface specification and the proxy's knowledge of the state of the team plan. The proxy's adherence to the STEAM rules ensures that its beliefs about the state of the team plan agree with those of its teammates. Thus, the proxy is sure that its domain agent will perform the requested task in synchronization with its teammates. The domain agent can then process the resulting task request, without necessarily being burdened with understanding the larger team context.

The domain agent returns any result it may produce to its Teamcore proxy. The proxy may then communicate the result to its teammates, as mandated by STEAM's coherence-preserving rules. Again, the domain agent need not know anything about the overall team context. In the case of a simple agent that provides responses to a fixed set of queries, it sees only a request from its Teamcore proxy that it processes and responds to, just as it would for any other individual client. However, the result of the domain agent's actions still produce the desired teamwide effects, since the Teamcore proxy forwards the result to those teammates to whom the new information is relevant.

The Teamcore proxies generate monitoring requests in a similar fashion, except that multiple team members may perform the same monitoring task without regard to any assigned roles within the organization. Thus, multiple proxies may send requests to their corresponding domain agents for more robust monitoring. One key interesting issue in this architecture is that it is often the domain agent, and not the Teamcore proxy, that has access to information relevant to the achievement, irrelevance, and unachievability of the team plans (e.g., an information-gathering agent can search a database for known threats to a team of helicopters). Yet, only the Teamcore proxy knows the current team plans, so the domain agent may not know what observations are relevant (e.g., threats to the helicopters are relevant), necessitating communication about monitoring. For each team plan, the Teamcore proxies already maintain the termination conditions — conditions that make the team plan achieved, irrelevant, or unachievable. Each Teamcore proxy also maintains a specification of what its domain agent can observe. Thus, if a domain agent can observe conditions that reflect the achievement, irrelevance, or unachievability of a team plan, then the Teamcore proxy automatically requests it to monitor any change in those conditions. The response from the domain agent may be communicated with other Teamcore proxies, through the usual STEAM procedures.

The Teamcore proxies can similarly translate STEAM's monitor and repair rules into appropriate messages for the domain agents. For instance, in the human collaboration domain, each proxy monitors its user's ability to attend the meeting on time, perhaps asking the user directly. If the user responds that s/he is unable to attend, the proxy follows the STEAM rules and automatically forwards this information to the rest of the team. If the user fills a critical role in the meeting plan (e.g., s/he is the presenter), then the team must repair the plan before proceeding. The proxies, again following the STEAM rules, first determine whether their users have the capability of taking on the role, perhaps by asking directly. Finally, the proxies follow the STEAM repair rules to fill the role with one of the users whom they determine to be capable and then notify the selected user.

Thus, the overall interface between a Teamcore proxy and its domain-level agent performs the following three tasks:

- If executing a team plan, α, which has termination conditions, send the conditions to the domain agent for monitoring
- If executing an individual plan that results in a task, send the task to be performed by the domain agent
- If the information sent by the domain agent matches the termination conditions of a team plan, use the STEAM algorithm as specified in Appendix A.

4. Team-oriented programming

Because the Teamcore proxies automatically handle much of the necessary coordination among the agents executing the desired tasks, the developer can specify those tasks at a more convenient abstract level through team-oriented programming. Section 4.1 describes how our JAVA-based TOPI helps a software developer in building an agent organization. Section 4.2 describes how KARMA can aid the developer by locating relevant agents and assisting in allocation of roles to such agents.

4.1. Constructing team plans and organization

The developer begins specifying an organization of interest via team-oriented programming, in which the developer specifies three key aspects of a team: a team organization hierarchy, a hierarchy of reactive team plans, and assignments of agents to plans. The team organization hierarchy consists of roles for individuals and for groups of agents. For example, Figure 4-a illustrates a portion of the organization hierarchy of the roles involved with the evacuation scenario (described in more detail in Section 2.1). Each leaf node corresponds to a role for an individual agent, while the internal noes correspond to (sub)teams of these roles. *Task Force* is thus the highest level team in this organization, while *Orders-Obtainer* is an individual role.

The second aspect of team-oriented programming is specifying a hierarchy of reactive team plans. While these reactive team plans are much like reactive plans for individual agents, the key difference is that the team plans explicitly express joint activities. The reactive team plans require that the developer specify the: (i) initiation conditions under which the plan is to be proposed; (ii) termination conditions under which the plan is to be ended, specifically, the conditions when the reactive team plan is achieved, irrelevant or unachievable; and (iii) team-level actions to execute as part of the plan. Figure 4-b shows an example from the evacuation scenario (please ignore the bracketed names for now). Here, high-level reactive team plans, such as **Evacuate**, typically decompose into other team plans, such as **Process-orders** (to interpret orders provided by a human commander). **Process-orders** is itself achieved via other sub-plans such as **Obtain-orders**. The precise detail of how to execute a leaf-level plan such as **Obtain-orders** is left unspecified — thus both simplifying the specification, and allowing for the use of different agents to execute this plan.

The software developer must also specify domain-specific plan-sequencing constraints on the execution of team plans. In the example of Figure 4, the plan **Landing-Zone-Maneuvers** has two subplans: **Mask-Observe** which involves observing the landing zone while hidden, and **Pickup** to pick people up from the landing zone. The developer must specify the domain-specific sequencing constraint that a subteam assigned to perform **Pickup** cannot do so until the other subteam assigned **Mask-Observe** has reached its observing locations.

The third aspect of team-oriented programming is the assignment of agents to plans. This is done by first assigning the roles in the organization hierarchy to plans and then assigning

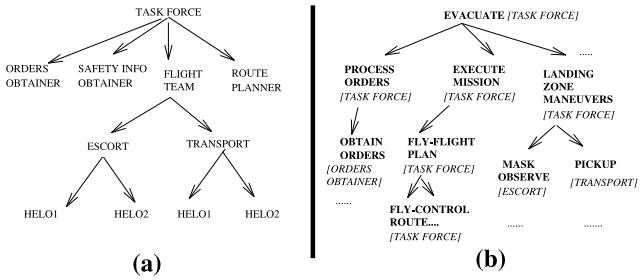

Figure 4. The evacuation scenario: (a) Partial organization hierarchy; (b) Partial team plan hierarchy.

agents to roles. Assigning only abstract roles rather than actual agents to plans provides a useful level of abstraction: new agents can be more quickly (re)assigned when needed. Figure 4-b shows the assignment of roles to the reactive plan hierarchy for the evacuation domain (in brackets adjacent to the plans). For instance, *Task Force* team is assigned to jointly perform **Evacuate**, while the individual *Orders-obtainer* role is assigned to the leaf-level **Obtain-orders** plan. Associated with such leaf-level plans are specifications of the requirements to perform the plan. For instance, for **Obtain-orders**, the requirement is to interact with a human. A role inherits the requirements from each plan that it is assigned to. Thus, the requirements of a role are the union of the requirements of all of its assigned plans. The assignment of agents to roles is discussed in the next subsection.

The real key here is what is *not* specified in the team-oriented program: details of how to realize the coordination specified, e.g., how members of *Task Force* should jointly execute **Evacuate**. Thus, for instance, the developer does not have to program any synchronization actions, because the coherence preservation rules of the proxies' STEAM module generate them automatically, as described in Section 3.1. Thus, during execution, synchronization actions among members of *Task Force* are automatically enforced, both with respect to the time of plan execution and the identity of the plan (i.e., all members will choose the same plan out of a set of multiple candidates). Similarly, there is no need to specify the coordination actions for coherently terminating reactive team plans; such actions are automatically executed by the proxies in accordance with the STEAM rules. Domain-specific plan-sequencing constraints, such as the one between **Mask-Observe** and **Pickup** discussed above, are also automatically enforced.

Likewise, the developer does not have to specify how team members should cover for each other in case of failures; rather, the proxies use the STEAM rules for monitoring and repair to automatically replace fallen teammates. The team-oriented programming phase automatically generates the required capabilities for each role in the organization, as well as the capabilities of each available agent. If an agent should fail during execution, the proxies can follow the STEAM rules to automatically find any available replacements for each of its roles based on these capability requirements.

Figure 5 shows a sample screenshot from TOPI used in programming the evacuation scenario, where the three panes correspond to the plan hierarchy (left pane), organization

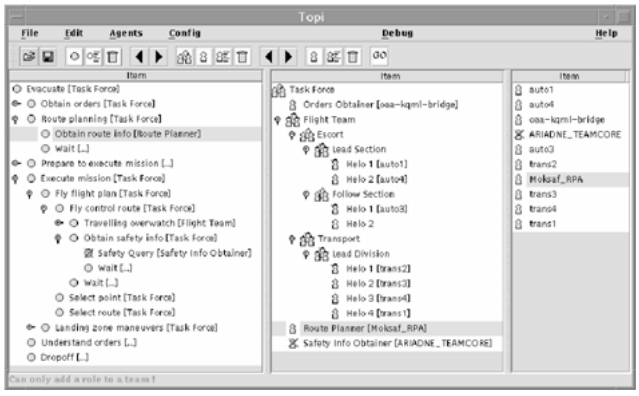

Figure 5. TOPI snapshot from generating team-oriented program for the evacuation scenario.

hierarchy (middle pane), and the domain agents (right pane). The left pane essentially reflects the diagram 4-b, e.g., *Task Force* has been assigned to execute **Evacuate**. Associated with each entity are its properties, e.g., associated with each plan are its coordination constraints, preconditions, assigned subteam, and so on.

4.2. Searching and assigning agents

As mentioned in the previous section, the team-oriented program assigns organizational roles to team plans. KARMA, our Knowledgeable Agent Resources Manager Assistant, derives requirements for these individual roles in the organization based on this assignment. KARMA searches for agents whose capabilities match these requirements. By limiting the search for available agents to just the organizational requirements, KARMA avoids overwhelming the software developer with a list of all available agents.

KARMA has multiple agent sources at its disposal: middle agents, local white pages directories of known agents, and other registry services. For instance, KARMA can query the AMatchMaker middle agent [7] by sending it a KQML message specifying an advertisement template. AMatchMaker returns descriptions of those agents whose advertised capabilities match the template. In addition, KARMA can search its own database of previously used agents or the local white pages service. KARMA is also interfaced with the agent registration and interconnection services provided by the Grid, as developed under the DARPA CoABS program [11].

Thus, from these different sources, KARMA compiles a list of relevant agents, and their properties, including address (host and port) and capabilities (some information, such as reliability, is available to KARMA from previous experience). From this list of relevant agents (in TOPI's right pane in Figure 5), the developer can assign agents to the roles in the specified organization. Once the developer has made an assignment, KARMA checks that the assignment is valid with respect to the plan requirements. This check is done in three steps. First, KARMA verifies that each agent has the capabilities required by its assigned role, where these requirements are derived from all of the role's assigned plans. Second, KARMA proceeds bottom-up through the team-plan hierarchy, recursively propagating the verification to the team operators as well. Third, KARMA verifies that basic constraints of an organization hierarchy are maintained, e.g., a child is not assigned higher than its parent in the plan hierarchy. The developer may also choose to allow KARMA to do the assignment automatically, which KARMA may do using a greedy approach, i.e., assigning to each role the first available agent that has all of the required capabilities.

Once the developer has used KARMA to fill in the required roles in the organization, the team-oriented programming phase is complete, and the Teamcore proxies can begin execution of the team plans, as described in Section 3.

5. Adapting to team member heterogeneity

This section focuses on the adaptive capabilities of the Teamcore proxies. These capabilities extend the proxies' fundamental coordination capabilities, as described in Section 3.

5.1. Dynamic plan alteration

The Teamcore proxies' general-purpose teamwork knowledge ensures that the execution of the team-oriented program is robust even in the face of complete failures of agents. However, the heterogeneity of the agents and the dynamics of the environment raise other types of dangerous situations as well. For instance, the developer may have designed the team-oriented program based on certain performance models of the agents involved, perhaps including response times, quality of solution, etc. In dynamic, real-world execution, it is possible that an agent's actual performance may deviate from its normal behavior. If the deviation is large enough, the developer may prefer a different team-oriented program than the original. In some cases, a domain agent may be alive but responding more slowly than it usually does. The team may not be able to wait indefinitely for the agent's response; on the other hand, the agent's response presumably has some value, so the team cannot simply give up whenever an agent takes longer than usual to respond. It is thus important that the Teamcore proxies be able to make runtime decisions about plan execution based on the performance of the domain agents.

We have designed the Teamcore architecture to allow dynamic decisions about whether to include or exclude any optional plans. Each Teamcore proxy has an initial specification of its domain agent's capabilities, including parameters such as response time (e.g., average, min/max response times are recorded from past runs). However, if the actual runtime performance of a domain agent greatly differs from expectations (e.g., so that the cost in agent response time greatly exceeds the benefits from its results), the Teamcore proxies together modify the optional plans and avoid using this particular domain agent.

Teamcore's plan representation for this decision-making is similar to that used by existing approaches to decision-theoretic planning [2]. Each plan has associated preconditions and postconditions, but unlike in a logical plan formalism, the plan specification here includes a probability distribution over the possible values of the postcondition state features, conditioned on the possible values of the precondition state features. For instance, in the evacuation scenario, the team plan for planning a route around obstacles (e.g., no-fly zones) is optional. A specification for the route-planning team plan could state that if there is a no-fly zone between the origin and destination points, then there is 50% chance that the resulting route will be twice as long as the straight-line distance, but it will also be safer because it avoids the no-fly zone. On the other hand, if there is *no* no-fly zone, then it has a 0% chance of being longer, since the resulting route will be exactly the straight-line route.

The Teamcore proxies can examine the various combinations of the optional plans, composing the probabilistic effects of the separate plans to determine the preconditions, postconditions, and distribution over the entire sequence. The developer can then specify a utility function over postconditions to represent the value of time, the cost of crossing a no-fly zone, etc. The Teamcore proxies can thus evaluate the expected utility over the possible sequences of team plans, where different sequences are generated by including or excluding optional plans. The Teamcore proxies begin executing a plan sequence that they determine (a priori) to be optimal given the probability of various outcomes over that sequence and the specified utility function.

However, the proxies can dynamically choose to omit optional plans if a particular domain agent's response time should deviate from the expected time cost. For instance, if

they had initially decided to include the route planning plan, but the route planner is taking longer than expected, the Teamcore proxies can compare their current plan sequence against alternate candidates (e.g., flying in straight-line, but unsafe, paths), taking into account the increased cost of the current response time. If the time cost outweighs the value of route planning, the Teamcore proxies can change sequences and skip the route-planning step, knowing that they are saving in the overall value of their execution.

In theory, to fully support such decision-theoretic evaluation, the developer must specify the value of executing each team plan in terms of its time cost and possible outcomes. We would then represent these as a probability distribution and utility function over possible states, with $\Pr(q_1|q_0, p)$ representing the probability of reaching state q_1 after executing plan p in state q_0, and with $U(q, p)$ representing the utility derived from executing plan p in state q. However, to ensure that the decision-theoretic evaluation remains practical, several approximations are used. First, the states here are not complete representations of the team state, but rather of only those features that are relevant to the optional plans. For instance, when evaluating route planning, the Teamcore proxies consider only the length of the route and whether that route crosses a no-fly zone prior to route-planning. Second, the decision-theoretic evaluation is done in terms of only the more abstract plans in our team-plan hierarchy, so developers need not provide detailed execution information about all plans, and Teamcore proxies need not engage in very detailed decision-theoretic evaluations. Third, for most plans, the derived utility is simply taken as a negative time cost. However, in the evacuation scenario, the team plans corresponding to helicopter flight have a value that increases when the helicopters reach their destination and that decreases with any time spent within a no-fly zone.

The probability distribution over outcomes allows the developer to capture the value of plans that have no inherent utility, but only gather and modify the team's information. For instance, the mission begins either in state q_{safe} with an overall route that does not cross any no-fly zones or in state q_{unsafe} with a route that does. The developer also specifies an initial distribution $\Pr(q)$ for the likelihood of these states. When executing the route-planning plan in state q_{unsafe}, the route planner creates a route around no-fly zones, so we enter state q_{safe} with a very high probability. The developer then provides the relative value of executing the *flight* plan in the two states through the utility function values $U(q_{safe}, \mathit{flight})$ and $U(q_{unsafe}, \mathit{flight})$.

The Teamcore proxies use this probability and utility information to select the sequence of plans $p_0, p_1, \ldots, p_n$ that maximizes their expected utility. In the evacuation scenario, there are only four such sequences, because only two team plans (out of the total of 40) are optional. They reevaluate their choice only when conditions (i.e., agent response times) have changed significantly. Thus, whenever a domain agent associated with either of these plans takes longer than usual to respond, its Teamcore proxy can find the optimal time (with respect to the specified utility function) for terminating the current information-gathering step and using a different plan sequence for the rest of the team-oriented program.

5.2. Adapting the level of autonomy

The central notion behind the Teamcore architecture is the use of proxies to create team players out of agents who may not know how to work together. However, agents differ in

their abilities to coordinate themselves. Many agents know how to do only their particular task, servicing requests without any regard for the larger multiagent context. At the opposite end of the spectrum, people can perform complicated coordination themselves and are thus less reliant on the Teamcore proxies for ensuring proper coordination. In fact, there are potential overlaps (and perhaps even conflicts) between the coordination actions taken by a Teamcore proxy and those preferred by the user it represents. People have strong preferences about what plans and actions they want their proxies to adopt on their behalf. Within the Teamcore architecture, proxies mediate all coordination, but it is important that they respect the preferences of the agents on whose behalf they act, while still fulfilling their responsibilities at the team level. Ideally, the Teamcore proxies should act autonomously in only those areas where there is no overlap, but they should certainly avoid performing coordination actions that their agents do not want taken.

A key challenge in integrating heterogeneous agents is that these conflicts may occur in different cases for different agents. For instance, in the human collaboration application (discussed in more detail in Section 2.2), a Teamcore proxy may commit its user to substitute for a missing discussion leader, knowing that its user is proficient in the discussion topics. However, the user may or may not want the proxy to autonomously make this commitment. The decision may vary from person to person, and may depend on many diverse factors. Conversely, though, restricting the proxy to always confirm its decision with the user is also undesirable, since it would then often overwhelm the user with confirmation requests for trivial decisions.

Thus, it is important that a proxy have the right level of autonomy. Yet, to avoid hand-tuning such autonomy for each person (or agent), it is critical for a proxy to automatically adapt its autonomy to a suitable level. We rely on a supervised learning approach (C4.5 [24]) using user feedback as the input data. Here, a key issue that contrasts our work with previous work on autonomy adaptation (e.g., [26]) is that the the level of autonomy is not only dependent on the individual but also on the the other agents being integrated. For instance, in the discussion leader example above, the number of other attendees and their state might be factors in the autonomy decision. Thus, in our approach we emphasize the use of knowledge about other team members (in addition to the preferences of the integrated agent) in using supervised learning techniques. Each proxy learns what decisions it can take autonomously, and what decision need be confirmed with the agent it represents *in the context of particular scenarios involving other agents*.

Specifically, the Teamcore proxies for people can make coordination and repair decisions autonomously to aid in team activities like meetings (e.g., the human collaboration domain). Eleven attributes are used in learning, some of which have been inspired by existing meeting scheduling systems, such as MeetingMaker,[3] which include meeting location, time, resources reserved, etc. However, other attributes describe the state of the other agents participating in the meeting—e.g., the number of persons attending and the most important member attending (in terms of the organizational hierarchy). These attributes are extracted from the user's schedule files, organizational charts, etc. In the training phase, a proxy suggests a coordination decision and a query as to whether the user would wish it to make such a decision autonomously. The proxies use C4.5 [24] to learn a decision tree from the interactions with their users. Each proxy thus learns what situations are appropriate for autonomous actions (decision tree d_{auto} : states $\rightarrow$ {autonomous, not autonomous}), as well as which autonomous actions are appropriate for those situations

(decision tree d_{act} : states → actions). In all other situations, the proxy defers to the user's choice before acting. The overall algorithm is as follows:

– If proxy faced with a coordination decision in state, s:

- If $d_{auto}(s)$ = autonomous:
  - Perform action $d_{act}(s)$
- Else: /* Agent should not act autonomously */
  - Ask user for input
  - Upon receiving user input, a, perform action a

Since the submission of this article, our further work into adjustable autonomy has resulted in changes and extensions to our approach [22, 27].

6. Evaluation

Our application of the Teamcore framework to the two application domains described in Section 2 has provided rich testbeds for evaluating the architecture's ability to successfully coordinate agents. Section 6.1 discusses the Teamcore architecture's ability to support robust coordination within the two domains. Section 6.2 discusses the advantages that team-oriented programming provided in integrating the agents in the two domains. Section 6.3 discusses the effectiveness of the architecture's adaptive capabilities.

6.1. Evaluation of robust execution of team plans

One key aspect of evaluation is robustness, one of the motivations for the teamwork foundations of Teamcore. In both application domains, we used our Teamcore framework to specify the necessary team-oriented program, assign domain agents, and launch their proxies, which then successfully and robustly executed the team-oriented program. The overall system runs successfully, continuing even in the face of software failures in individual agents; instead, the Teamcore proxies of the remaining agents try to substitute another agent with relevant expertise if necessary (the maintenance-and-repair rules of STEAM) and/or show graceful degradation. For instance, if the route planner or its Teamcore proxy were to suddenly crash, the system does not halt. Instead, it reverts back to using straight-line paths.

In the evacuation domain, we built an agent organization from the 11 available domain agents (listed in Section 2.1, with eight separate helicopter pilot agents). The team-oriented program contained 43 reactive team plans. KARMA located the domain agents based on the team-oriented program and the specified organization hierarchy (although, this particular research collaboration was pre-arranged with the other groups). The Teamcore proxies then successfully executed the team-oriented program. In other words, they communicated with each other when appropriate and generated the correct tasking and monitoring requests for the domain agents. To demonstrate the robustness of the resulting agent organizations, the Teamcore-based system for evacuation rehearsal has often been demonstrated live outside our laboratory.

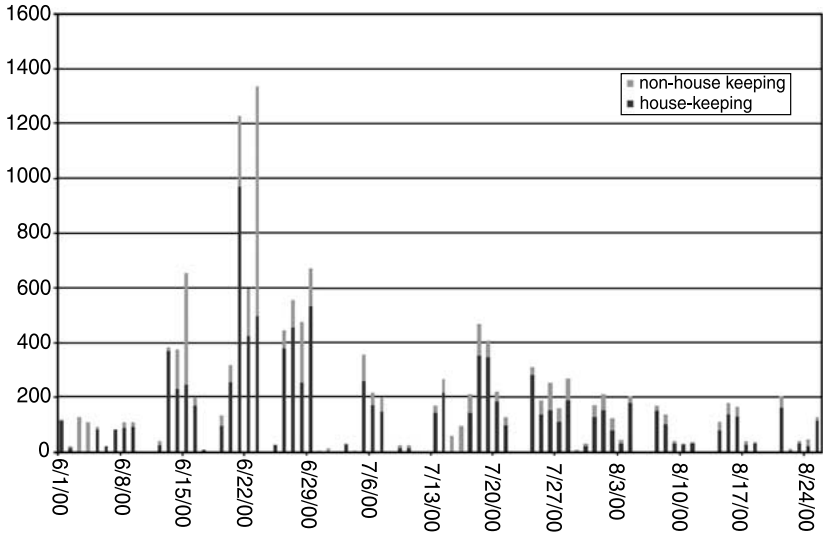

Figure 6. Number of daily coordination messages exchanged by proxies over three-month period.

In the human collaboration domain, as of the writing of this paper, the proxies have run 24 hours/day, 7 days/week for three months.[4] We have designed, implemented, and deployed proxies to coordinate the activities for fifteen members of our research division. Here, the team-oriented program contains 15 reactive team plans, but each team member has multiple instantiations of many of these plans. For instance, there are several different **Successful-meeting** team plans active in parallel, one for each meeting the user has scheduled in the coming week. Figure 6 plots the number of daily messages exchanged by the proxies while coordinating these plans. The size of the counts demonstrates the large amount of coordination actions necessary in managing all of the plans, while the high variability of the daily count illustrates the dynamic nature of the domain.

The distributed Teamcore architecture is well-suited to this domain, since people can maintain control of their own proxies, rather than centralizing the control and meeting information. Each proxy serves its user's interests in dealing with team-level activities. Thus, the proxy reasons about the user's willingness to accept joint activities and assigned roles within those activities. This reasoning requires that the proxies monitor their users' state and make decisions (possibly without any user input) about what they should report about that state to the team in the service of team goals.

The proxy often needs to interact with its user, whether to inform the user of a change in the status of a joint activity (e.g., a rescheduled meeting) or to ask for information before acting on a joint activity (e.g., whether the user wants a meeting delayed). As described in Section 2.2, the proxy exploits the user's workstation displays, any available PDAs, and email to third parties. The workstation display provides a simple Graphical User Interface (GUI) that allows the user to view what joint activities the proxy is currently monitoring. The GUI allows the user to initiate certain actions without waiting for the proxy to make them autonomously. The proxy displays the user's schedule and any informative messages on the PDA. The proxy can also pop up a dialog box on the PDA for feedback.

6.2. *Evaluation of team-oriented programming*

One key dimension of the benefits of Teamcore proxy's in-built teamwork capabilities is in the abstraction provided by team-oriented programming. One key alternative to such an in-built teamwork model is reproducing all of Teamcore's capabilities via domain-specific coordination plans. In such a domain-specific implementation, about 10 separate domain-specific coordination plans would be required for each of the 40 team plans in Teamcore [32]. That is, we would require potentially hundreds of domain-specific coordination plans to reproduce Teamcore's capabilities to coordinate among each other for this domain alone. In contrast, with Teamcore, no coordination plans were written for inter-Teamcore communication. Instead, such communications occurred automatically from the specifications of team plans. Thus, it would appear that Teamcore has significantly alleviated the coding effort for coordination plans.

The Teamcore proxies' use of STEAM's selectivity in communication provides the additional benefit of automatically minimizing the amount of communication needed for proper coordination. Figure 7 shows the number of messages exchanged over time in different runs. The X axis measures the time elapsed while the Y axis shows the cumulative number of messages exchanged on a *log scale*. The "normal" run shows the number of messages typically exchanged among the Teamcore proxies for the evacuation scenario with time. The key here is that these approximately 100 messages are automatically generated by the Teamcore proxies. The "cautious" run shows the number of messages (approximately 1000) exchanged among the Teamcore proxies without the decision-theoretic communication selectivity in Teamcore, illustrating both the overhead reduction via such reasoning and the difficulty of hand-coding coordination (simple hand-coded coordination may lead to significant overheads). Finally, the "failure" run shows the messages exchanged among the proxies if the Ariadne agent were to crash unexpectedly (in the course of a "normal" run). To compensate for such failure, there is an initial increase in the total messages; but once the proxies compensate for the failure, fewer messages are exchanged, so that the total messages in the "failure" and "normal" runs is roughly the same.

Team-oriented programming also simplifies organizational modifications. Our framework appears to facilitate changes to the team, at least compared with the alternative of

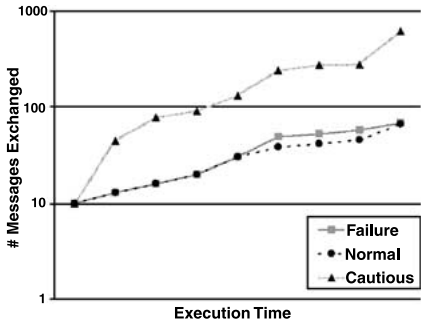

Figure 7. Comparison of messages exchanged.

domain-specific coordination. For instance, the route planner was the last addition to the team. It required few modifications to the team-oriented program. To extend the organizational hierarchy, we simply added the route planner as a member of *Task Force*. We then added the **Process-Routes** branch of the plan hierarchy to allow for route planning. This branch involves very simple plans where the Teamcore proxy submits a request for planning a particular route, waits for the reply by the route planner, and then communicates the new route to the other team members according to STEAM's coherence-preserving rules. Again, no new coordination rules were required. It is similarly easy to modify the organization in the human collaboration example, where members join and leave, and teams form and dissolve.

6.3. Evaluation of adaptation

6.3.1. Evaluation of dynamic plan alteration. We can also evaluate the benefit of Teamcore's runtime plan modification capabilities (see Section 5.1). Figure 8 shows the results of varying Ariadne's response times on the time of the overall mission execution. In the evacuation plan, Ariadne provides information about missile locations along a particular route. If there are missiles present, the proxies instruct the helicopters to fly at a higher altitude to be out of range. The organization could save itself the time involved with querying Ariadne by simply having the helicopters *always* fly at the higher safe altitude. However, the helicopters fly slower at the higher altitude, so the query is sometimes worthwhile, depending on Ariadne's response time. In Figure 8, we can see that when Ariadne's response time exceeds 15 seconds, the cost of the query outweighs the value of the information. In such cases, the Teamcore proxies with the plan-alteration capability skip the query to save in overall execution time, while an inflexible team cannot achieve such savings.

6.3.2. Evaluation of adjustable autonomy. In evaluating the proxy's adjustable autonomy component, we began with data that came from "interviews" of the users on hypothetical situations that called for potential rescheduling. Here, we used actual meeting data from 58 meetings taken from the users' scheduling programs. We asked the users

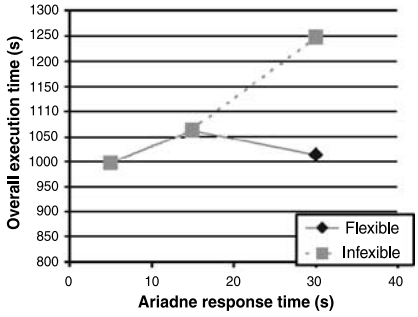

Figure 8. Comparison of flexible vs. inflexible execution of the evacuation plan, with respect to Ariadne's response time and value.

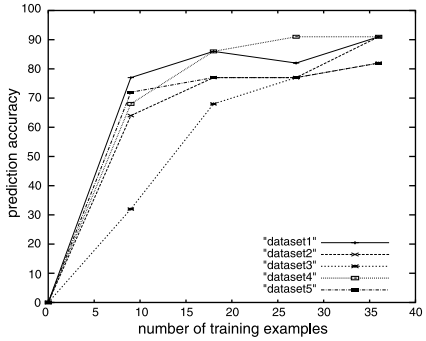

Figure 9. Progress of proxy's learning user's preferences for choice decision during hypothetical meeting situations.

what actions they would want their proxies to take if they became delayed and whether they would want their proxies to confirm decisions with them before acting. We extracted five different data sets by randomly selecting 36 meetings for training data and 22 for test data (i.e., random sub-sampling holdout). Figure 9 displays the accuracy of the agent proxies' predictions of action choices, plotted against the number of examples used to train the agent (out of the 36 training examples), for the five different data sets. For each data set, we observe that the prediction accuracy would usually reach 91%. However, using more than 36 examples did not produce any further noticeable improvements.

Figure 10 presents the test results from two weeks of real-world execution of proxies for two of the more experienced users (20 decision examples for user1, 10 for user2). During their execution, the proxies continuously reinvoke C4.5 to learn from new data as it becomes available. The agent proxies apply the learned rules to predict new action choices as required. Figure 10 shows that user1's proxy accurately predicts action choices after only 10 meetings, while user2's proxy does not show any improvement in accuracy. The proxy for user2 had fewer decision examples at its disposal, so we hope that the system will improve its prediction as more examples become available. As for the learning results for the autonomy decision, user1 was willing to give complete autonomy to the proxy, which seems reasonable given the accuracy of the proxy's predictions. On the other hand, user2 wanted the proxy to confirm its predictions with him in all cases, which again seems reasonable given the inaccuracy of the proxy's predictions of his action choices. Given the

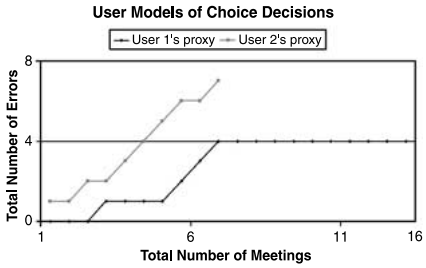

Figure 10. Progress of proxy's learning user's preferences for choice decision during real meeting situations.

extreme nature of these user preferences, the system had no trouble learning the correct decision exactly.

7. Related work

This paper unifies and extends our previous publications on Teamcore, specifically conference publications [23, 35] and a magazine article [34]. This journal article unifies the various contents of these previous publications within a single framework. It also provides detailed algorithms for the separate components, which was not possible in the previous publications given their space limitations.

Section 1 briefly discusses Teamcore's relationship with centralized integration architectures such as OAA [20]. The Adaptive Agent Architecture (AAA) [19] provides a more distributed extension to OAA, allowing for teamwork among the facilitators. However, AAA does not provide teamwork among the agents themselves, thus limiting the robustness it can guarantee for the integrated system. Another related system, the RETSINA multi-agent infrastructure framework [30], is based on three different types of interacting agents: (i) interface agents; (ii) task agents; and (iii) information agents. Middle agents allow these various agents to locate each other. This effort appears quite complementary to Teamcore. Indeed, as Section 4.2 discusses, KARMA can use RETSINA middle agents for locating relevant agents, while infrastructural teamwork in Teamcore may enable the different RETSINA agents to work in teams.

COLLAGEN [25] models dialogue between a user and an agent — a form of joint activity — based on the SharedPlans [10] model of joint action. COLLAGEN has been previously compared to STEAM, and, like Teamcore, it provides a fairly clean separation between the teamwork layer and the problem-solving layer of the agent. However, COLLAGEN targets wrapping only a single agent for collaboration with a user, so that the issue of constructing and programming a team of agents and humans is not relevant.

Other than COLLAGEN, few other agent integration frameworks explicitly address the possibility of integrating people within the multiagent system. The rest of this section describes some integration frameworks that have been applied only to software agents, but that are still relevant to our Teamcore architecture. Tidhar [36, 37] used the term ''team-oriented programming'' to describe a conceptual framework for specifying team behaviors based on mutual beliefs and joint plans, coupled with organizational structures. This framework forms the basis of an implementation based on the dMars agent architecture [38]. In Tidhar's framework, the organizational hierarchy ensures that only appropriate agents (e.g., team leaders) fill specific roles offering certain authority or privilege. Tidhar describes how one can automatically unfold team plans into plans for individual agents containing communicative acts that ensure rudimentary coordination. His framework also addressed the issue of team selection [39] — team selection matches the ''skills'' required for executing a team plan against agents that have those skills. While many of the features of Tidhar et al.'s conceptual and implemented frameworks are important in the context of Teamcore, the critical issue of agent reuse, particularly involving heterogeneous (non-dMars) agents, is not given much attention. Thus, proxies with monitoring, tasking, and plan alteration capabilities and an agent resources manager such as KARMA for locating

agents are novel in our framework. Furthermore, Teamcore's flexibility of reorganization and communication selectivity (through STEAM) does not seem to be part of the abstract team layer of Tidhar's framework.

Jennings's GRATE* [15] work also uses a teamwork module, one that has been previously compared to STEAM. GRATE* implements a model of cooperation based on the joint intentions framework, similarly used by STEAM. Each agent has its own *cooperation level* module that negotiates involvement in a joint task and maintains information about its own and other agents' involvement in joint goals. Regarding the specific issue of agent reuse, GRATE* separates the teamwork layer from the individual problem-solving layer of an agent. However, Teamcore's STEAM module allows teamwork to a deeper level than the single joint goal and plan in GRATE*. The more complex nature of the teams and team tasks in Teamcore has led us to explicitly focus on team-oriented programming and to explore several novel issues (e.g., automatic generation of monitoring conditions) that GRATE* does not address. STEAM also provides capabilities for role substitution in repairing team activity, a capability not available in GRATE*. Furthermore, GRATE* also does not address the issues of building team-oriented programs to specify agent organizations and KARMA-like agent resources manager to aid in building and monitoring such programs.

The ADEPT architecture for modeling business processes [17] allows a more flexible, hierarchical team organization than GRATE*. ADEPT consists of multiple *agencies*, each containing a *responsible agent*, which handles communication and interaction with other agencies. Various responsible agents maintain each agency's ''capabilities'', avoiding the use of a central facilitator or broker. A task is ''contracted out'' to an agency that has the capabilities to perform that task. As with GRATE*, ADEPT provides a fairly clean interface between the individual task-achieving agents and the social level. However, the ADEPT framework does not seem to address the issue of agent reuse directly, although the architecture itself could potentially incorporate heterogeneous agents. Also, ADEPT does not provide an explicit model of teamwork, such as that based on joint plans/intentions; instead, the basis of collaboration seems more closely related to the notion of *social commitment* [3].

Singh has proposed an abstract framework for coordinating heterogeneous agents [29]. Singh's model represents planned activity via finite-state automata (abstracting away the internal workings of the agents), where transitions represent external actions or events. The coordination service maintains knowledge of individual agents' actions as well as the overall joint plan and, upon receiving a request to perform an action, informs the appropriate agents as to whether an intended action should be executed, delayed, or omitted so as to fit with the joint activity of other agents. Singh's model does not address many of the issues of teamwork; however, it provides a potentially useful tool which could augment the joint plan framework of Teamcore with a language for specifying flexible, coordinated interactions at an abstract level.

Like the STEAM rule module within Teamcore, the COOL coordination framework [1] also focuses on general-purpose coordination by relying on obligations among agents. However, it explicitly rejects the notion of joint goals and joint commitments. It would appear that individual commitments in COOL would be inadequate in addressing some teamwork phenomena, but further work is necessary in understanding the relationship between COOL and Teamcore.

8. Summary

The two application domains tackled in this work, as well as most other real-world domains, present unique agent integration challenges of heterogeneity of agents (both software and human), number of the joint activities, complexity of the properly coordinated behavior, etc. While no previous agent integration architecture has yet resolved all of these challenges simultaneously, the Teamcore architecture takes a significant step forward. Teamcore's success in its two widely disparate application domains demonstrates the power and generality of the overall framework and provides strong evidence for its underlying hypotheses.

The primary hypothesis behind the Teamcore architecture and a key lesson learned from its success is that an agent integration infrastructure based on sound principles of agent coordination can automate robust coordination among distributed, heterogeneous agents. The Teamcore proxies' teamwork model proved sufficient in enabling robust coordination among the agents in both domains. The proxies' ability to reuse the same general-purpose rules to accomplish this robustness, despite the vast differences between the two domains, demonstrates the effectiveness of this teamwork knowledge.

In addition, by using the separate Teamcore proxies to perform the coordination, we are able to incorporate the domain agents without modifying them. This is especially important in an open environment, where our access to the internals of the agents is often minimal. It is also difficult (or at least undesirable) to modify the internals of humans. Thus, the proxies' teamwork knowledge succeeded in coordinating agents that were essentially black boxes and that spanned the entire range of coordination capabilities, from the team-ready humans to the software agents completely incapable of coordination.

Teamwork also provides a useful layer of abstraction for coordinating heterogeneous agents. As described in Section 4, once the agents become good team players (through their proxies), we are free to design at the level of team-oriented programming. We have constructed organizations using team members that covered the spectrum of agenthood, from databases capable of answering KQML queries, to people capable of multiple, parallel tasks and modes of interaction. Fortunately, in designing these organizations, one can ignore the details about the agents themselves. The developer instead thinks in terms of relevant joint activities: which agents are working together, what is the task they are performing, and who plays what roles. The proxies make the abstract specification of these joint activities operational by using their teamwork model to fill in the details about what and when messages would actually go among the agents.

Teamwork provides a sound basis for a coordination architecture, but it is important that the teamwork be flexible, to maximize the architecture's applicability across multiple domains. For instance, in the evacuation domain, where the domain agents had no coordination knowledge themselves, the Teamcore proxies were completely responsible for synchronizing the actions of the agents. The proxies would then be sure to communicate before initiating critical plans when miscoordinated execution among the various domain agents was sufficiently likely and costly. On the other hand, in the human collaboration domain, the domain agents are people who are already very capable of coordinating themselves. Thus, there was less of a burden on the Teamcore proxies to

synchronize certain team plans, which then had a low cost of miscoordinated execution in the team-oriented program.

In addition to flexibility in coordination specification, the coordination architecture must provide flexibility in execution. Another key lesson learned from Teamcore's success is that adaptive capabilities provide a necessary degree of robustness when faced with dynamic, heterogeneous agents we do not completely know a priori. Without this adaptation, the proxies would not have been able to properly operate on behalf of the diverse members of our research group or in the face of the dynamic agent performance in the evacuation domain. The Teamcore architecture demonstrated the value of its flexibility in successfully supporting the two vastly different domains described in this paper.

In conclusion, the Teamcore architecture provides a novel means for integrating distributed, heterogeneous software agents and humans. Its use of a general-purpose teamwork model supports abstract specification of coordination behavior, robust execution of that behavior, and adaptation to world dynamics and heterogeneity. In addition, the proxy-based architecture supports coordination in an open environment where agent modifications may not always be possible. The success of the Teamcore architecture in its two application domains encourages us to continue expanding its capabilities and applications. We will continue running the proxies for the human collaboration domain, but we also plan to expand the number and types of team activities that they manage. We also plan to integrate more and more of our fellow researchers, as well as additional software agents, into the agent organization. As the size and complexity of the organization grows, the architecture will have to address new issues deriving from this scale-up (e.g., conflicts between team activities). We believe that the architecture's success will extend to many more real-world domains as well, providing a powerful framework for agent integration beyond that currently possible with existing technology.

Appendix A. Detailed TEAMCORE specification

The pseudo-code described below follows the description of STEAM provided in this article. It is based on execution of hierarchical team reactive plans (as specified in a team-oriented program). All reactive-plans in the hierarchy execute in parallel, and hence the "in parallel" construct. The comments in the pseudo code are enclosed in /* */. The terminology is first described below, to clarify the pseudo-code

- *Execute-Team-Oriented-Program*(α, Θ, **C**, {$\rho 1$, $\rho 2, \ldots, \rho n$}) denotes the execution of a team reactive-plan α, by a team Θ, given the context of the current intention hierarchy **C**, and with parameters $\rho 1$, $\rho 2 \ldots \rho n$.
- $[\alpha]_\Theta$ denotes the team Θ's joint intention to execute α.
- **EU** denotes computation to determine expected utility of an action, e.g., communication action.
- **status**($[\alpha]_\Theta$, *STATUS-OF*-α) denotes the status of the joint intention $[\alpha]_\Theta$, whether it is mutually believed to be achieved, unachievable or irrelevant.
- **satisfies** (Achievement-conditions(α), *f*) denotes that the fact *f* satisfies the achievement conditions of the team reactive-plan α; similarly with respect to unachievability and irrelevancy conditions.

- *Communicate*(terminate-jpg(α), *f*, Θ) denotes communication to the team Θ to terminate Θ's joint commitment to α, due to the fact f.
- Update-state (**team-state**(Θ), *f*) denotes the updating of the team state of Θ with the fact f.
- Update-status($[\alpha]_\Theta$) denotes the updating of the team reactive-plan α with its current status of achievement, unachievability or irrelevancy.
- **Agent**(α) is the individual agent or team executing reactive-plan α.
- **actions**(α) denote the actions of the reactive-plan α.
- **teamtype**(ψ) is a test of whether the agent ψ is a team or just one individual.
- **self**(ψ) is a test of whether the agent ψ denotes self.
- **agent-status-change**(μ) denotes change in the role performance capability of agent or subteam μ.
- *Execute-individual-reactive-plan*(ψ, self, **C**, {ρ1, ρ2,...,ρn}) denotes the execution of an individual reactive-plan ψ by self, given the context of the current intention hierarchy **C**, and with parameters ρ1, ρ2...ρn.
- *Execute-reactive-plan-to-send-task*(ψ, self, **C**, {ρ1, ρ2,...,ρn}) denotes the execution of an individual reactive-plan ψ by self, given the context of the current intention hierarchy **C**, and with parameters ρ1, ρ2...ρn, to send a task to the domain-level agent.
- *Execute-reactive-plan-to-send-monitoring-conditions*(ψ, self, **C**, {Achievement-conditions(α), Unachievability-conditions(α), Irrelevance-conditions(α)}) denotes the execution of an individual reactive-plan ψ by self, given the context of the current intention hierarchy **C**, and with parameters of the termination conditions of α, to send all the termination conditions as monitoring conditions to the domain-level agent.

A.1. Team-oriented program execution

Execute-Team-Oriented-Program(α, Θ, **C**, {ρ1, ρ2,...,ρn})
{

1. Is **EU**(communicate) > **EU** (not-communicate) *execute establish joint commitment* protocol;
2. establish joint-intention $[\alpha]_\Theta$;
3. While NOT(**status**($[\alpha]_\Theta$, **Achieved**) ∨ **status**($[\alpha]_\Theta$, **Unachievable**) ∨ **status**($[\alpha]_\Theta$, **Irrelevant**)) Do
   {
   a) *Execute-reactive-plan-to-send-monitoring-conditions*(βi, *self*, α/**C**, **C**, {Achievement-conditions(α), Unachievability-conditions(α), Irrelevance-conditions(α)})
   b) *Execute-reactive-plan-to-receive-monitoring-results*(βi, *self*, α/**C**, ρ1...);
   c) if (**satisfies** (Achievement-conditions(α), *f*) ∨ **satisfies** (Unachievability-conditions(α), *f*) ∨ **satisfies** (Irrelevance-conditions(α), *f*))
   /* This is the case where fact f obtained by receiving monitoring results from domain agent is found to satisfy the termination condition of α. */
   {
   i) if **EU**(communicate) > **EU**(not-communicate) propose-reactive-plan *Communicate*(terminate-jpg(α), *f*, Θ) with high priority;
   ii) if no other higher priority reactive-plan, in parallel

Execute-individual-reactive-plan(*Communicate*(terminate-jpg(α), *f*, Θ), *self*, α/**C**, {ρ1, ρ2,...});

iii) Update-state (**team-state**(Θ), *f*);

iv) Update-status($[\alpha]_\Theta$);

}

d) if **agent-status-change**(μ), where $\mu \in \Theta$

{

i) Evaluate role-monitoring constraints;

ii) if role-monitoring constraint failure **cf** such that (**satisfies** (Unachievability-conditions(α), **cf**) then update-status($[\alpha]_\Theta$);

}

e) if receive communication of terminate-jpg(α) and fact *f*

{

if (**satisfies**(Achievement-conditions(α), *f*) $\vee$ **satisfies** (Unachievability-conditions(α), *f*) $\vee$ **satisfies** (Irrelevance-conditions(α), *f*))

{

i) Update-state (**team-state**(Θ), *f*);

ii) Update-status($[\alpha]_\Theta$);

}

}

f) Update-state(**team-state**(Θ), **actions**(α));

/* execute domain-specific actions to modify team state of Θ */

g) if children reactive-plan β1, β2,...βn of α proposed as candidates

{

i) βi $\leftarrow$ select-best{β1...βn};

ii) if (**teamtype**(**Agent**(βi)) $\wedge$ (Θ = **Agent**(βi))) then in parallel *Execute-Team-Oriented-Program*(βi, Θ, α/**C**, {ρ1, ρ2...});

iii) if (**teamtype(Agent(**βi**))** $\wedge$ **(Agent(**βi**))** $\subset$ Θ) then in parallel

{

A) *Execute-Team-Oriented-Program*(βi, **Agent**(βi), α/**C**, {ρ1, ρ2...});

B) Instantiate role-monitoring constraints;

}

iv) if **self**(**Agent**(βi)) then in parallel

{

A) *Execute-reactive-plan-to-send-task*(βi, *self*, α/**C**, ρ1...);

B) Instantiate role-monitoring constraints;

}

}

} /* End while statement in 4 */

4. terminate joint intention $[\alpha]_\Theta$; /* Terminating this joint intention also leads to termination of children intentions, and a message being sent to domain agent to terminate any activities initiated due to this team plan */
5. if **status**($[\alpha]_\Theta$, **Unachievable**)

{

if (α != *Repair*) /* If α is not itself Repair */

{
Execute-Team-Oriented-Program(*Repair*, Θ, **C**, {α, cause-of-unachievability,...}) /* Repair enables recovery by substitution of another team member, for instance. Cause-of-unachievability, passed as a parameter to Repair, may be role-monitoring constraint violation as in case 4b, or the domain-specific unachievability conditions. */
} else {
Execute-Team-Oriented-Program(*Complete-Failure*, Θ, **C**, {α, cause-of-unachievability, ...}) /* If Repair is itself unachievable, complete-failure results */
}
}
} /* end procedure execute-Team-Oriented-Program */

Appendix B. STEAM sample rule

The sample rules described below follow the description of STEAM provided in this article, and essentially help encode the algorithm described in Appendix A. The rules, as with the algorithm in Appendix A, are based on execution of hierarchical reactive-plans, or reactive plans. While the sample rules below are described in simplified if-then form, the actual rules are encoded in Soar.

SAMPLE:RULE:CREATE-COMMUNICATIVE-GOAL-ON-ACHIEVED
/* This rule focuses on generating a communicative goal if an agent's private state contains a belief that satisfies the achievement condition of a team reactive-plan $[OP]_{\Theta}$. */
IF
agent νi's private state contains a fact F
AND
fact F matches an achievement condition AC of a team reactive-plan $[OP]_{\Theta}$
AND
fact F is not currently mutually believed
AND
a communicative goal for F is not already generated
THEN
create possible communicative goal CG to communicate fact F to team Θ to terminate $[OP]_{\Theta}$.

SAMPLE:RULE:CREATE-COMMUNICATIVE-GOAL-ON-UNACHIEVABLE
/* This rule is similar to the one above. */
IF
agent νi's private state contains a fact F
AND
fact F matches an unachievability condition UC of a team reactive-plan $[OP]_{\Theta}$
AND
fact F is not currently mutually believed

AND
a communicative goal for F is not already generated
THEN
create possible communicative goal CG to communicate fact F to team Θ to terminate $[OP]_\Theta$.

SAMPLE:RULE:DECISION-ON-COMMUNICATION
/* This rule makes the communication decision. */
IF
CG is a possible communicative goal to communicate fact F to team Θ to terminate $[OP]_\Theta$
AND
Estimated value of non-communication for CG is **medium**
AND
Estimated value of communication for CG is **low**
THEN
post CG as a communicative goal to communicate fact F to team Θ to terminate $[OP]_\Theta$

SAMPLE:RULE:MONITOR-UNACHIEVABILITY:AND-COMBINATION
/* This rule checks for unachievability of role-monitoring constraints involving an AND-combination. */
IF
A current joint intention $[OP]_\Theta$ involves an AND-combination
AND
νi is a member performing role to execute sub-reactive-plan op
AND
no other member νj is also performing role to execute sub-reactive-plan op
AND
νi cannot perform role
THEN
Current joint intention $[OP]_\Theta$ is unachievable, due to a critical role failure of νi in performing op

Acknowledgments

The following people contributed to various phases in the development of the Teamcore architecture and the two domain applications presented here: Nicolas Chauvat, Abhimanyu Das, Lei Ding, Neelesh Gokhale, Gal A. Kaminka, Haeyoung Lee, Pragnesh J. Modi, and Paul Scerri. This research was supported by DARPA Award no. F30602-98-2-0108 and managed by the AFRL/Rome research site. We thank Hans Chalupsky, Phil Cohen, Yolanda Gil, Craig Knoblock, Steve Minton, and Katia Sycara for contributing agents used in the work described here.

Notes

1. Teamcore derives its name from its encapsulation of "core team reasoning" as discussed later.
2. Section 4 provides more details about the contents of the team-oriented program component of the proxy.
3. http://www.meetingmaker6.com
4. These experimental runs are occasionally interrupted for bug fixes and enhancements.

References

1. M. Barbuceanu and M. Fox, "The architecture of an agent building shell," in M. Wooldridge, J. Muller, and M. Tambe, (eds.), *Intelligent Agents, Volume II: Lecture Notes in Artificial Intelligence 1037*, Springer-Verlag: Heidelberg, Germany, 1996.
2. J. Blythe, "Planning with external events," in *Proc. of the Conference on Uncertainty in Artif. Intell.*, pp. 94–101, 1994.
3. C. Castelfranchi, "Commitments: From individual intentions to groups and organizations," in *Proceedings of International Conference on Multi-Agent Systems*, pp. 41–48, 1995.
4. H. Chalupsky, Y. Gil, C. A. Knoblock, K. Lerman, J. Oh, D. V. Pynadath, T. A. Russ, and M. Tambe, "Electric elves: Applying agent technology to support human organizations," To appear in *Proceedings of the Innovative Applications of Artificial Intelligence Conference*, 2001.
5. P. R. Cohen, M. Johnston, D. McGee, S. Oviatt, J. Pittman, I. Smith, L. Chen, and J. Clow, "QuickSet: Multimodal interaction for distributed applications," in *Proceedings of the Fifth Annual International Multimodal Conference (Multimedia '97)*, pp. 31–40, 1997.
6. P. R. Cohen and H. J. Levesque, "Teamwork," *Nous*, vol. 35, 1991.
7. K. Decker, S. Sycara, and M. Williamson, "Middle-agents for the internet," in *Proc. of the Internat'l Joint Conf. on Artif. Intell.*, 1997.
8. T. Finin, R. Fritzson, D. McKay, and R. McEntire, "KQML as an agent communication language," in *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM '94)*, 1994.
9. B. Grosz and S. Kraus, "Collaborative plans for complex group actions," *Artificial Intelligence*, vol. 86, pp. 269–358, 1996a.
10. B. Grosz and S. Kraus, "Collaborative plans for complex group actions," *Artificial Intelligence*, vol. 86, pp. 269–358, 1996b.
11. J. Hendler and R. Metzeger, "Putting it all together – the control of agent-based systems program," *IEEE Intell. Systems and Their Applications*, vol. 14, 1999.
12. M. N. Huhns, "Networking embedded agents," *IEEE Internet Computing*, vol. 3, pp. 91–93, 1999.
13. M. N. Huhns and M. P. Singh, "All agents are not created equal," *IEEE Internet Comp.*, vol. 2, pp. 94–96, 1998.
14. N. Jennings, "Controlling cooperative problem solving in industrial multi-agent systems using joint intentions," *Artificial Intelligence*, vol. 75, 1995a.
15. N. Jennings, "Controlling cooperative problem solving in industrial multi-agent systems using joint intentions," *Artificial Intelligence*, vol. 75, pp. 195–240, 1995b.
16. N. Jennings, "Agent-based computing: Promise and perils," in *Proceedings of the International Joint Conference on Artificial Intelligence*, 1999.
17. N. Jennings, T. J. Norman, and P. Faratin, "ADEPT: An agent-based approach to business process management," *ACM SIGMOD Record*, vol. 27, no. 4, pp. 32–39, 1998.
18. C. A. Knoblock, S. Minton, J. L. Ambite, N. Ashish, P. J. Modi, I. Muslea, A. G. Philpot, and S. Tejada, "Modeling web sources for information integration," in *Proceedings of the National Conference on Artificial Intelligence*, 1998.
19. S. Kumar, P. R. Cohen, and H. J. Levesque, "The adaptive agent architecture: Achieving fault-tolerance using persistent broker teams," in *Proceedings of the International Conference on MultiAgent Systems*, pp. 159–166, 2000.
20. D. L. Martin, A. J. Cheyer, and D. B. Moran, "The open agent architecture: A framework for building distributed software systems," *Applied Artif. Intell.*, vol. 13, nos. 1–2, pp. 92–128, 1999.

21. A. Newell, *Unified Theories of Cognition*, Harvard University Press: Cambridge, MA, 1990.
22. D. V. Pynadath and M. Tambe, "Revisiting Asimov's first law: A response to the call to Arms," in *Proceedings of the IJCAI-01 Workshop on Autonomy, Delegation, and Control: Interacting with Autonomous Agents*, 2001.
23. D. V., Pynadath, M. Tambe, N. Chauvat, and L. Cavedon, "Toward team-oriented programming," in *Proceedings of the Agents, Theories, Architectures and Languages (ATAL '99) Workshop (to be published in Springer Verlag "Intelligent Agents V")*, pp. 77–91,1999.
24. J. R. Quinlan, *C4.5: Programs for Machine Learning*, Morgan Kaufmann: San Mateo, CA, 1993.
25. C. Rich and C. Sidner, "COLLAGEN: When agents collaborate with people," in *Proceedings of the International Conference on Autonomous Agents (Agents '97)*, 1997.
26. S. Rogers, C. Fiechter, and P. Langley, "An adaptive interactive agent for route advice," in *Third International Conference on Autonomous Agents*, Seattle, WA, 1999.
27. P. Scerri, D. V. Pynadath, and M. Tambe, "Adjustable autonomy in real-world multi-agent environments," in *Proceedings of the Conference on Autonomous Agents*, pp. 300–307, 2001.
28. Y. Shoham, "Agent-oriented programming," *Artificial Intelligence*, vol. 60, no. 1, pp. 51–92, 1993.
29. M. P. Singh, "A customizable coordination service for autonomous agents," in *Proceedings of the 4th International Workshop on Agent Theories, Architectures and Languages (ATAL'97)*, 1997.
30. K. Sycara, K. Decker, A. Pannu, M. Williamson, and D. Zeng, "Distributed intelligent agents," *IEEE Expert*, vol. 11, pp. 36–46, 1996.
31. C. Szyperski, *Component Software: Beyond Object-oriented Programming*, Addison Wesley: Menlo Park, CA, 1999.
32. M. Tambe, "Towards flexible teamwork," *Journal of Artificial Intelligence Research*, vol. 7, pp. 83–124, 1997.
33. M. Tambe, J. Adibi, Y. Alonaizon, A. Erdem, G. Kaminka, S. Marsella, and I. Muslea, "Building agent teams using an explicit teamwork model and learning," *Artif. Intell.*, vol. 110, no. 2, 1999.
34. M. Tambe, D. V. Pynadath, and N. Chauvat, "Building dynamic agent organizations in cyberspace," *IEEE Internet Computing*, vol. 4, no. 2, 2000a.
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 International Conference on MultiAgent Systems*, pp. 301–308, 2000b.
36. G. Tidhar, "Team-oriented programming: preliminary report," Technical Report 41, Australian Artificial Intelligence Institute, 1993a.
37. G. Tidhar, "Team-oriented programming: social structures," Technical Report 47, Australian Artif. Intell. Inst., 1993b.
38. G. Tidhar, C. Heinze, and M. Selvestrel, "Flying together: Modelling air mission teams," *Journal of Applied Intelligence*, vol. 8, no. 3, 1998.
39. G. Tidhar, A. S. Rao, and E. A. Sonenberg, "Guided team selection," in *Proceedings of the Second International Conference on Multi-Agent Systems*, 1996.