AFRL-IF-RS-TR-2002-253
Final Technical Report
September 2002

TEAMCORE PROJECT CONTROL OF AGENT-BASED SYSTEMS (COABS) PROGRAM

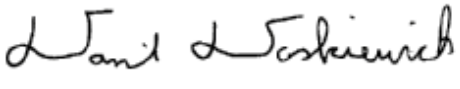
University of Southern California

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. G335

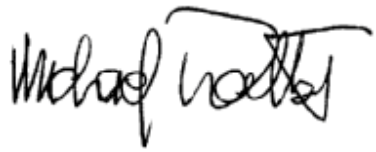
AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2002-253 has been reviewed and is approved for publication.

APPROVED: 

DANIEL E. DASKIEWICH
Project Engineer

FOR THE DIRECTOR: 

MICHAEL TALBERT, Maj., USAF, Technical Advisor
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	SEPTEMBER 2002	Final May 98 – May 02

4. TITLE AND SUBTITLE
TEAMCORE PROJECT CONTROL OF AGENT-BASED SYSTEMS (COABS) PROGRAM

5. FUNDING NUMBERS
C - F30602-98-2-0108
PE - 63760E
PR - AGEN
TA - T0
WU - 13

6. AUTHOR(S)
Milind Tambe

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Southern California
Information Sciences Institute
4676 Admiralty Way
Marina Del Rey California 90292

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington Virginia 22203-1714

AFRL/IFTB
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2002-253

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Daniel E. Daskiewich/IFTB/(315) 330-7731/ Daniel.Daskiewich@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

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. Systern 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. Integrating agents to perform real-world tasks in a large-scale system remains difficult. As part of DARPA's Control of Agent-Based Systems (CoABS) program, the Teamcore project addressed this challenge of agent integration by focusing on general-purpose teamwork capabilities. Based on 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. This report describes key contributions of the Tearncore project in areas of teamwork theory, team monitoring, adjustable autonomy and team oriented programming.

14. SUBJECT TERMS
Software Agents, Adjustable Autonomy, Teamwork Theory, Programming

15. NUMBER OF PAGES
40

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

1. Overview 1

2. Team-Oriented Programming 3

2.1 Constructing Team Plans and Organization 4

2.2 Searching and Assigning Agents 6

2.3 STEAM: Making Heterogeneous Agents Team Ready 7

2.4 Teamcore's Interface with Domain Agents 8

2.5 Application: CoABS TIE 1 (Mission Rehearsal) 10

3. Electric Elves 13

3.1 Coordination of Component Agents 14

3.2 Agent Interactions with Human Users 15

3.3 The E-Elves in Daily Use 16

3.4 Travel Elves 20

3.4.1 Email Agents 22
3.4.2 Palm Pilot Agents 22
3.4.3 Fax Agents 22
3.4.4 HTML Agents 22
3.4.5 WAP Agents 23
3.4.6 Speech Agents 23
3.4.7 Display Agents 23
3.4.8 Fax Server Agent 24

3.5 Travel Elves Deployment 24

4. Adjustable Autonomy 25

4.1 Transfer-of-control Strategies 26

4.2 Evaluation of AA Strategies 27

5. Monitoring Agent Teams 28

6. Teamwork Theory 30

7. Technology Transfer 33

7.1 Psychological Operations 33

7.2 Disaster Rescue 33

8. Personnel 33

9. Publications 33

Figure 1: Teamcore framework: Teamcore proxies for heterogeneous domain agents. 3
Figure 2: Evacuation scenario: (a) Partial organization hierarchy; (b) Partial plan hierarchy. 4
Figure 3: TOPI snapshot from generating team-oriented program for the evacuation scenario. 6
Figure 4: Reasoning components of a Teamcore proxy and interactions with domain agent. 9
Figure 5: ModSAF view of simulated helicopters. 11
Figure 6: Quickset view of evacuation scenario. 12
Figure 7: PDA (Palm VII) with GPS device for wireless, handheld communication between proxy and user. 14
Figure 8: Electric Elves System Architecture 15
Figure 9: Sample dialog box by which Friday asks for user input into meeting delay decisions. 16
Figure 10: Number of daily coordination messages exchanged by proxies over a seven-month period. 17
Figure 11: Monitored vs. delayed meetings per user. 18
Figure 12: Meetings delayed autonomously vs. by hand. 18
Figure 13: Sample screen shot from auction, used by Electric Elves to assign the presenter at a research group meeting. 19
Figure 14: Agent architecture for user proxy subteam as deployed in Travel Elves. 21
Figure 15: WAP-enabled cellular phone for wireless communication between proxy and user. 23
Figure 16: Web-based interface for administration of Electric Elves user agents and device teams. 25
Figure 17: Helicopter scenario used as example domain for COM-MTDPs. 32

1. Overview

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. 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.

Unfortunately, integrating agents to perform real-world tasks in a large-scale system remains difficult. As part of DARPA's Control of Agent-Based Systems (CoABS) program, our Teamcore project addressed this challenge of agent integration by focusing on general-purpose teamwork capabilities. Based on 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.

This report describes the five key contributions of our Teamcore project:

1. **Team-Oriented Programming (TOP):** Our first contribution is the Teamcore architecture for agent integration. It enables teamwork among agents with no coordination capabilities, and it establishes and automates consistent teamwork among agents with some coordination capabilities. Teamcore accomplishes this coordination by making 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 address the need for robust execution by automatically generating the required coordination actions for the agents they represent. Through *team-oriented programming*, a developer specifies a hierarchical organization and its goals and plans, abstracting away from coordination details. Thus, team-oriented programming provides a level of abstraction that can be used on top of previous approaches to agent-oriented programming. We used our Teamcore architecture to successfully address the challenges of agent integration in two application domains: simulated rehearsal of a military evacuation mission (the CoABS TIE 1) and facilitation of human collaboration (the Electric Elves).
2. **Electric Elves:** The second application domain of our Teamcore architecture, the Electric Elves, represents the first real-world, long-term deployment of a multi-agent system. Since June 1, 2000, teams of software agents have aided researchers at USC/ISI in accomplishing their tasks, facilitating the organization's coherent functioning and rapid response to crises, while reducing the burden on humans. Tied to individual user workstations, fax machines, voice, mobile devices such as cell phones and palm pilots, the Electric Elves system has assisted us in routine tasks, such as rescheduling meetings, selecting presenters for research meetings, tracking people's locations, organizing lunch meetings, etc.
3. **Adjustable Autonomy:** Central to the success of the Electric Elves system was our novel approach to adjustable autonomy in a team setting. Adjustable autonomy refers to

agents' dynamically varying their own autonomy, transferring decision-making control to other entities (typically human users) in key situations. Previous work has provided several different techniques to address this question, but often focused on individual agent-human interactions. Unfortunately, domains (like Electric Elves) requiring collaboration between teams of agents and humans reveal two key shortcomings of these previous techniques. To remedy these problems, we devised a novel approach to adjustable autonomy, based on the notion of *transfer of control strategy*. A transfer of control strategy consists of a conditional sequence of two types of actions: (i) actions to transfer decision-making control (e.g., from the agent to the user or vice versa) and (ii) actions to change an agent's pre-specified coordination constraints with others, aimed at minimizing miscoordination costs. The goal is for high-quality individual decisions to be made with minimal disruption to the coordination of the team. We operationalized these strategies using Markov Decision Processes to select the optimal strategy given an uncertain environment and costs to individuals and teams.

4. **Monitoring Agent Teams:** Having deployed multi-agent systems in the CoABS Tie and in the Electric Elves, we then needed methods for on-line monitoring of such teams of cooperating agents, e.g., for visualization, or performance tracking. However, in monitoring deployed teams, we often cannot rely on the agents to always communicate their state to the monitoring system. We developed a non-intrusive approach to monitoring by *overhearing*, where the monitored team's state is inferred (via plan recognition) from team-members' *routine* communications, exchanged as part of their coordinated task execution, and observed (overheard) by the monitoring system. Key challenges in this approach include the demanding run-time requirements of monitoring, the scarceness of observations (increasing monitoring uncertainty), and the need to scale-up monitoring to address potentially large teams. To address these challenges, we developed a set of complementary novel techniques that exploited knowledge of the social structures and procedures in the monitored team.

5. **Teamwork Theory:** In addition to our practical successes in multiagent teamwork, we also achieved fundamental breakthroughs at the theoretical level. Despite the significant progress in multiagent teamwork, we found that existing teamwork research did not address the *optimality* of its prescriptions nor the *complexity* of the teamwork problem. Without a characterization of the optimality-complexity tradeoffs, it is impossible to determine whether the assumptions and approximations made by a particular theory gain enough efficiency to justify the losses in overall performance. To provide a tool for use by multiagent researchers in evaluating this tradeoff, we developed a unified framework, the *COMmunicative Multiagent Team Decision Problem (COM-MTDP)*. The COM-MTDP model combines and extends existing multiagent theories, such as decentralized partially observable Markov decision processes and economic team theory. In addition to their generality of representation, COM-MTDPs also support the analysis of both the optimality of team performance and the computational complexity of the agents' decision problem. In analyzing complexity, we derived breakdown of the computational complexity of constructing optimal teams under various classes of problem domains, along the dimensions of observability and communication cost. In analyzing optimality, we exploited the COM-MTDP's ability to encode existing teamwork theories and models to encode two instantiations of joint intentions theory taken from the literature. Furthermore, the COM-MTDP model provides a basis for the development of novel coordination algorithms. We derived a domain-independent criterion for optimal communication and provided a

comparative analysis of the two joint intentions instantiations with respect to this optimal policy. We have implemented a reusable, domain-independent software package based on COM-MTDPs to analyze teamwork coordination strategies, and we demonstrated its use by encoding and evaluating the two joint intentions strategies within an example domain.

The following sections describe each of these contributions in order.

2. Team-Oriented Programming

Figure 1 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 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, our Knowledgeable Agent Resources Manager Assistant, 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 1**). 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.

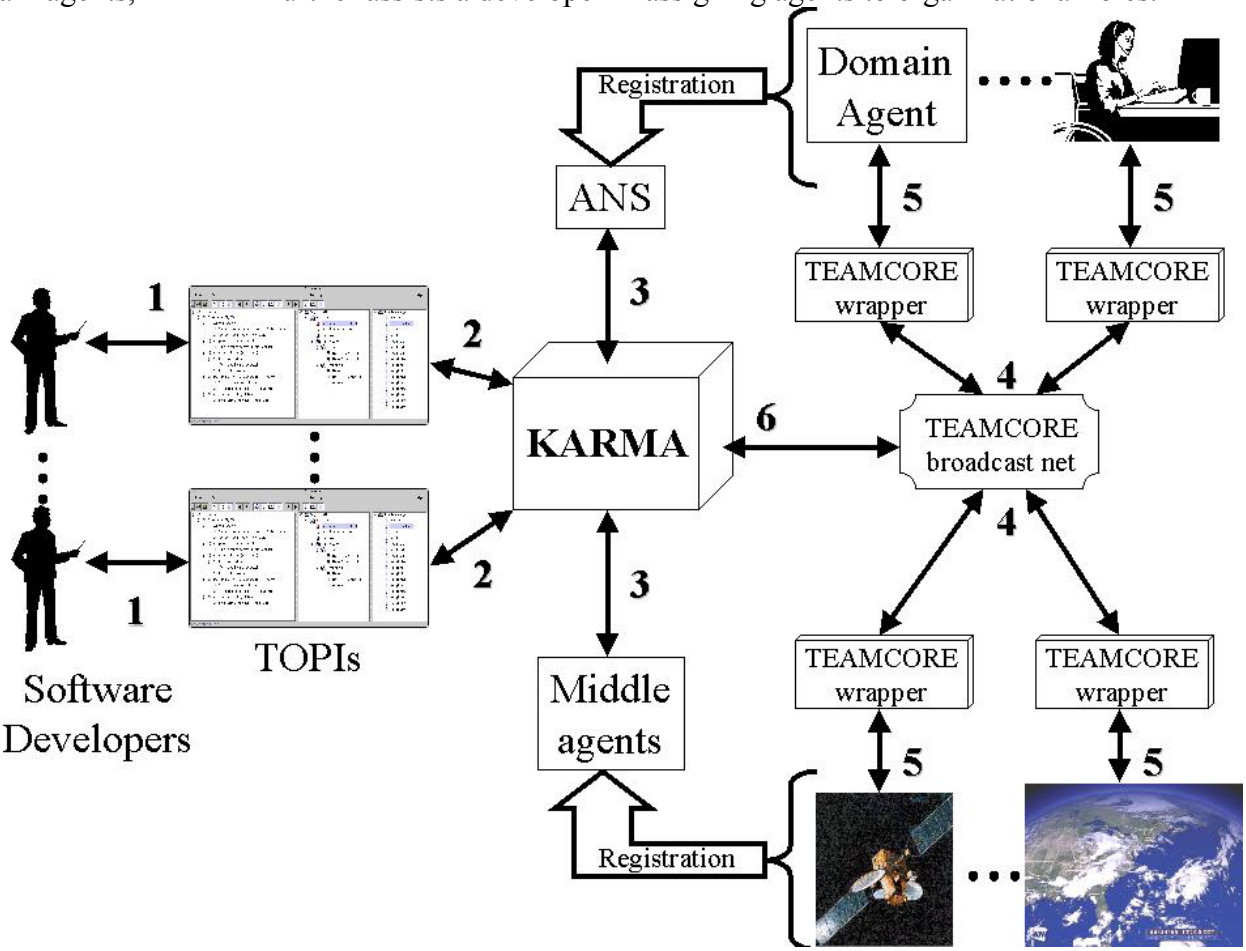


Figure 1: Teamcore framework: Teamcore proxies for heterogeneous domain agents.

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.

Section 2.1 describes stage 1 in more detail. The communication in stage 2 occurs via straightforward inter-process communication. Sections 2.2, 2.3, and 2.4 describe stages 3, 4, and 5 in more detail. We leave discussion of stage 6 to Section 5. Section 2.5 describes the application of the overall architecture to the CoABS TIE 1.

2.1 Constructing Team Plans and Organization

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. 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. Using the CoABS TIE 1 mission rehearsal for illustration, Figure 2-a illustrates a portion of the organization hierarchy of the roles involved with the evacuation scenario (described in more detail in Section 2.5). Each leaf node corresponds to a role for an individual agent, while the internal nodes 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.

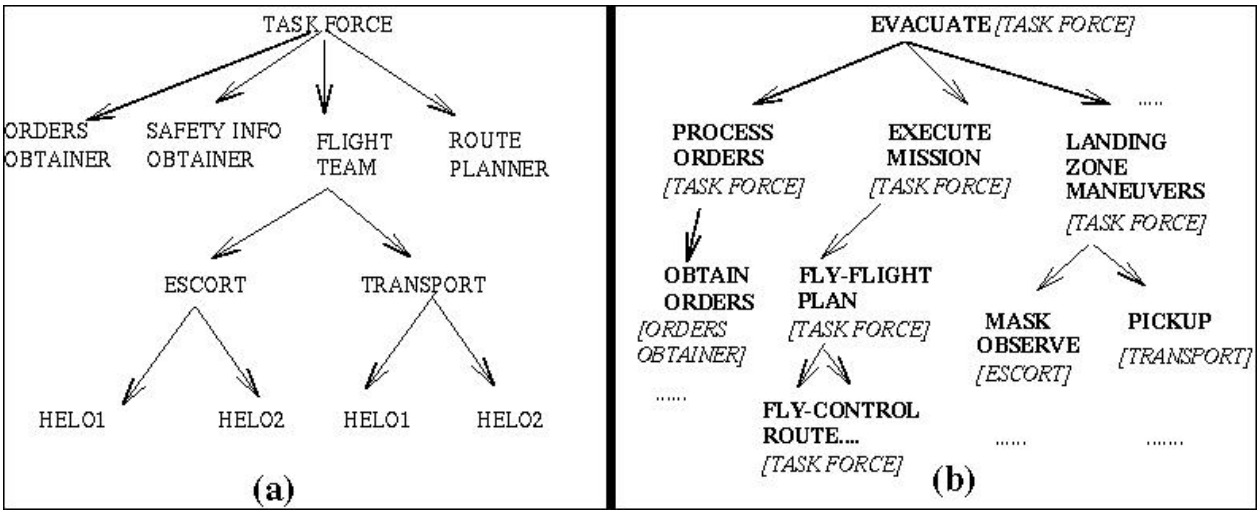


Figure 2: Evacuation scenario: (a) Partial organization hierarchy; (b) Partial plan hierarchy.

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 2-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 2, 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 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 2-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' Shell for Teamwork (STEAM) module generate them automatically, as described in Section 2.3. 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; the proxies automatically execute such actions 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 3 shows a sample screenshot from TOPI used in programming the evacuation scenario, where the three panes correspond to the plan hierarchy (left pane), organization hierarchy (middle pane), and the domain agents (right pane). The left pane essentially reflects the diagram Figure 2-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.

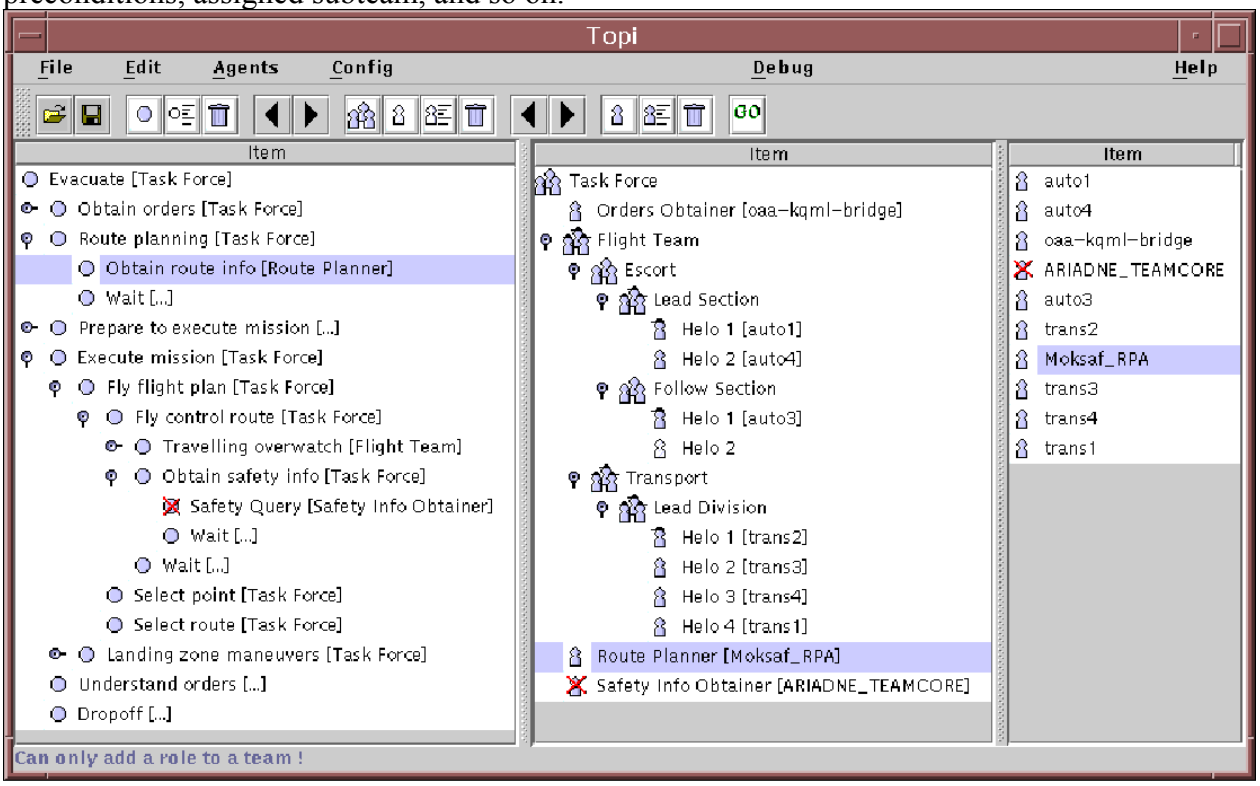


Figure 3: TOPI snapshot from generating team-oriented program for the evacuation scenario.

2.2 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 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 CoABS Grid.

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 3), 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.

2.3 STEAM: Making Heterogeneous Agents Team Ready

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. To ensure robust execution, the Teamcore architecture transforms agents of all types into a set of consistent team players. We achieve this team readiness among heterogeneous agents by providing each agent with a Teamcore proxy. The distributed Teamcore proxies, based on the Soar rule-based integrated agent architecture, execute their joint plans in a distributed fashion and coordinate as a team during this execution.

Each proxy contains the STEAM domain-independent teamwork module, responsible for Teamcore's teamwork reasoning. The STEAM algorithm is specified in detail on-line: (`http://www.isi.edu/teamcore/COM-MTDP/`). We have implemented the algorithm using production rules in Soar. 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 should 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.

2.4 Teamcore's Interface with Domain Agents

In previous work, STEAM resided directly in the domain agent's knowledge base, which is often difficult (if not impossible) to implement in an open, heterogeneous 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 4. 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.

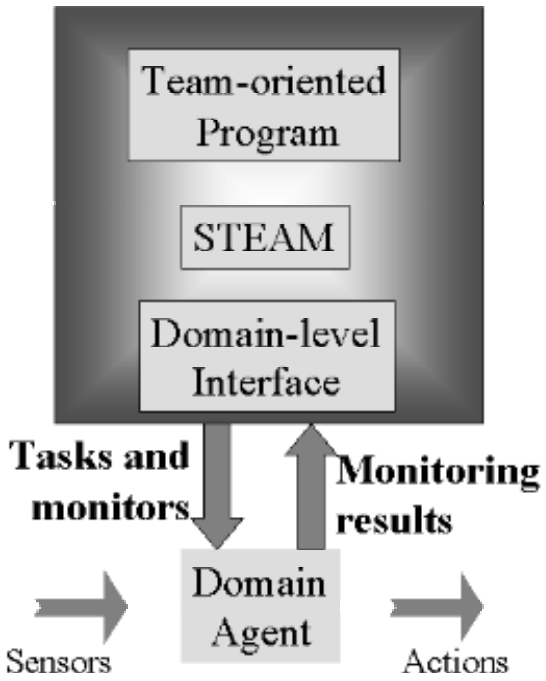


Figure 4: Reasoning components of a Teamcore proxy and interactions with domain agent.

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.

2.5 Application: CoABS TIE 1 (Mission 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]
• **Route planner:** (from Sycara et al., Carnegie-Mellon University) Retsina path planner for aircraft [C++, Windows NT]
• **Ariadne:** (from Minton et al., USC Information Sciences Institute) Database engine for dynamic threats [Lisp, Unix]
• **Helicopter pilots:** (from Tambe, USC Information Sciences Institute) Pilot agents for simulated helicopters [Soar, Unix]

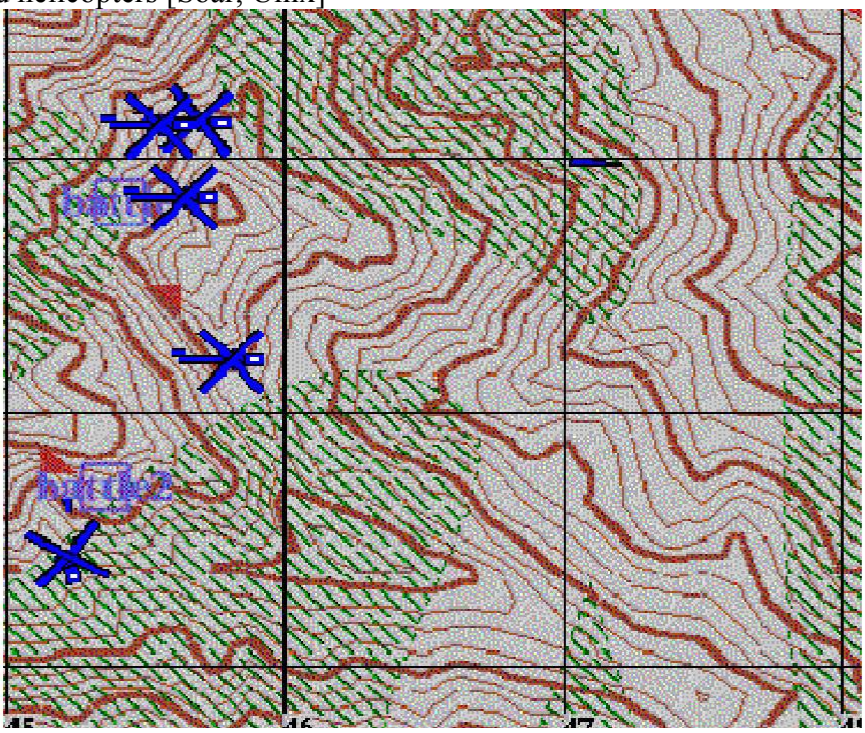


Figure 5: ModSAF view of simulated helicopters.

Figure 6: Quickset view of evacuation scenario.

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.

The Teamcore-based teams successfully met the challenge by generating correct task and monitoring requests, coordinating the domain-level agents' behavior to successfully accomplish the evacuation scenario. However, we are also interested in the effort involved in encoding and modifying agents' teamwork capabilities — comparing the effort with Teamcore against the alternatives. If we reproduced all of Teamcore's capabilities by providing the domain-level agents with special-case coordination plans, we would then require an ability to modify the code of the domain-level agents. In addition, we would also have to re-code the coordination plans in the languages used by each of the domain-level agents.

A better alternative would use domain-specific wrapper agents, but each of the 18 team operators in Teamcore would still require separate domain-specific communication plans for coordination — two plans each to signal commitments (request and confirm) and one to signal termination of commitments. Furthermore, reproducing Teamcore's selective communication would require additional special cases. In the extreme case, each combination of values for communication costs and rewards could require a separate special case operator (18×3× total combinations, already more than a hundred). Of course, we could economize all such special cases by discovering generalizations, but Teamcore already encodes such generalizations.

The Teamcore specification greatly facilitated modifications to the team as well. For instance, the route planner was the last addition to the team. To extend the organizational

hierarchy, we simply added the route planner as a member of Big-Team and added the **Process-Routes** branch of the goal hierarchy. This branch involves very simple goals where the Teamcore agent 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. The Teamcore teamwork model already supported most of this communication. Thus, the bulk of the coding effort for adding the route planner came in the specification of its message formats and task constraints.

3. Electric Elves

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 7). 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.

Figure 7: PDA (Palm VII) with GPS device for wireless, handheld communication between proxy and user.

After an initial exploration, this domain evolved into the "Electric Elves" [15,1] and has been reported in the popular press as well (USA Today, http: //www.usatoday.com).

3.1 Coordination of Component Agents

The Electric Elves system spans a wide variety of component technologies and languages, communication protocols as well as operating system platforms. **Figure 8** shows the components of the current version of Electric Elves. Teamcore agents are written in Python and Soar (which is written in C), Ariadne wrappers are written in C++, the PHOSPHORUS capability matcher is written in Common-Lisp and the PowerLoom interest matcher is written in STELLA which translates into Java. The agents are distributed across SunOS 5.7, Windows NT, Windows 2000 and Linux platforms, and use TCP/IP, HTTP and the Lockheed KQML API to handle specialized communication needs. To solve the communication-level integration problem, we are using the CoABS Grid technology.

The diverse agents in Electric Elves must work together to accomplish the complex tasks of the whole system. For instance, to plan a lunch meeting, the interest matcher finds a list of potential attendees, the Friday of each potential attendee decides whether s/he will attend, the capability matcher identifies dietary restrictions of the confirmed attendees, and the reservation site wrapper identifies possible restaurants and makes the final reservation. In addition to low-level communication issues, there is the complicated problem of getting all these agents to work together as a team. Each of these agents must execute its part in coordination with the others, so that it performs its tasks at the correct time and sends the results to the agents who need them.

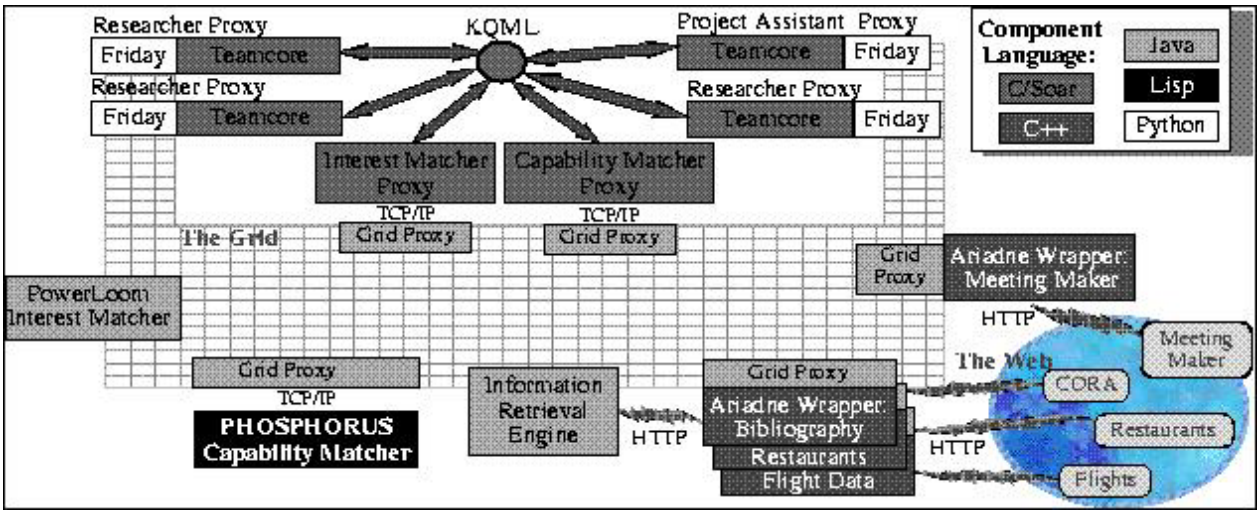


Figure 8: Electric Elves System Architecture

In Electric Elves, the agents coordinate using Teamcore. Each and every agent in the Electric Elves organization (Fridays, matchers, wrappers) has an associated Teamcore proxy that records its membership in various teams and active commitments made to these teams. Given an abstract specification of the organization and its plans, the Teamcore proxies *automatically* execute the necessary coordination tasks. They form joint commitments to team plans such as holding meetings, hosting and meeting with visitors, arranging lunch, etc. Teamcore proxies also communicate amongst themselves to ensure coherent and robust plan execution. The Teamcore proxies automatically substitute for missing roles (e.g., if the presenter is absent from the meeting) and inform each other of critical factors affecting a team plan. Finally, they communicate with their corresponding agents to monitor the agents' ability to fulfill commitments (e.g., asking Friday to monitor its user's attendance of a meeting) and to inform the agents of changes to those commitments (e.g., notifying Friday of a meeting rescheduling).

3.2 Agent Interactions with Human Users

Electric Elves agents must often take actions on behalf of the human users. Specifically, a user's agent proxy (named "Friday" after Robinson Crusoe's servant and companion) can take autonomous actions to coordinate collaborative activities (e.g., meetings). Friday's decision making on behalf of a person naturally leads to the issue of *adjustable autonomy*. An agent has the option of acting with full autonomy (e.g., delaying a meeting, volunteering the user to give a presentation, ordering a meal). On the other hand, it may act without autonomy, instead asking its user what to do. Clearly, the more decisions that Friday makes autonomously, the more time and effort it saves its user. Yet, given the high uncertainty in Friday's knowledge of its user's state and preferences, it could potentially make very costly mistakes while acting autonomously. For example, it may order an expensive dinner when the user is not hungry, or volunteer a busy user to give a presentation. Thus, each Friday must make intelligent decisions about when to consult its user and when to act autonomously. Section 4 describes our approach to adjustable autonomy in more detail.

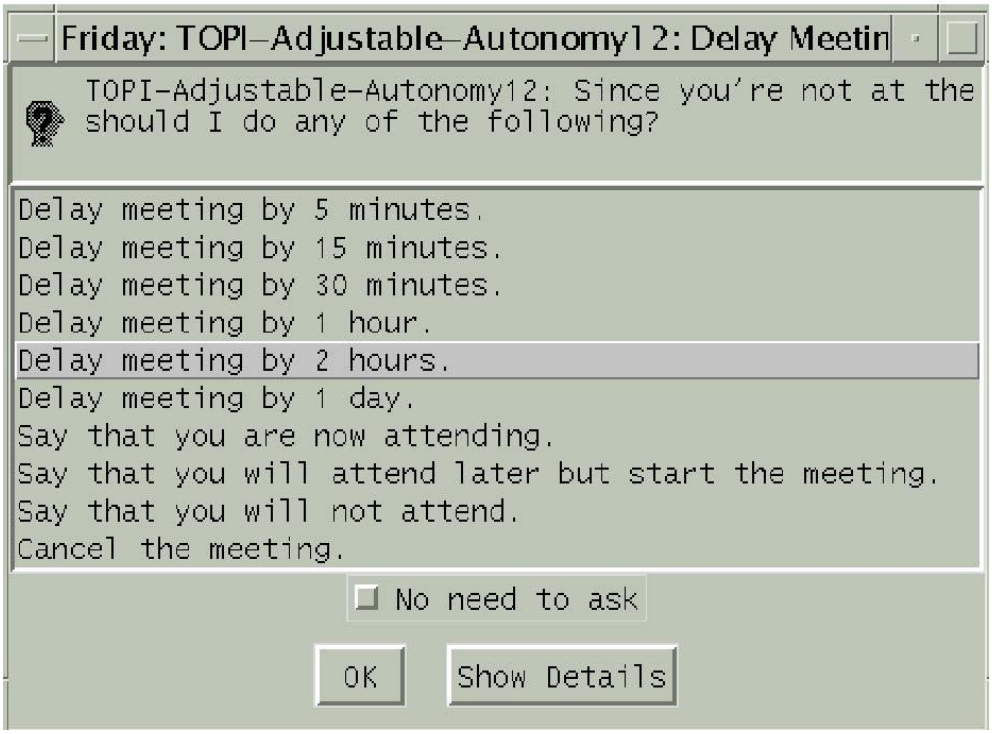


Figure 9: Sample dialog box by which Friday asks for user input into meeting delay decisions.

3.3 The E-Elves in Daily Use

The E-Elves system was heavily used by ten users in a research group at ISI, between June 2000 and December 2000.[1] The Friday agents ran continuously, around the clock, seven days a week. The exact number of agents running varied over the period of execution, with usually five to ten Friday agents for individual users, a capability matcher (with proxy), and an interest matcher (with proxy). Occasionally, temporary Friday agents operated on behalf of special guests or other short-term visitors.

The general effectiveness of the E-Elves is shown by several observations. During the E-Elves' operation, the group members exchanged very few email messages to announce meeting delays. Instead, Fridays autonomously informed users of delays, thus reducing the overhead of waiting for delayed members. Second, the overhead of sending emails to recruit and announce a presenter for research meetings was assumed by agent-run auctions. Third, a web page, where Friday agents post their users' location, was commonly used to avoid the overhead of trying to track users down manually. Fourth, mobile devices kept users informed remotely of changes in their schedules, while also enabling them to remotely delay meetings, volunteer for presentations, order meals, etc. Users began relying on Friday so heavily to order lunch that one local "Subway" restaurant owner even suggested: *"...more and more computers are getting to order food...so we might have to think about marketing to them!"*

[1]The user base of the system was greatly reduced after this period due to personnel relocations and student graduations, but it remains in use to this date with a smaller number of users.

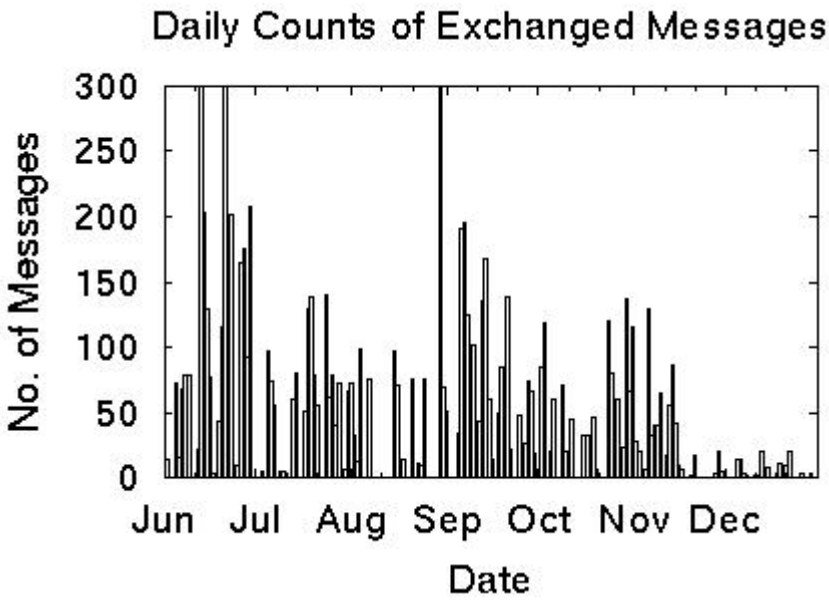


Figure 10: Number of daily coordination messages exchanged by proxies over a seven-month period.

Figure 10 plots the number of daily messages exchanged by the Fridays over seven months (June through December, 2000). The size of the daily counts reflects the large amount of coordination necessary to manage various activities, while the high variability illustrates the dynamic nature of the domain (note the low periods during vacations and final exams). Figure 11 illustrates the number of meetings monitored for each user. Over the seven months, nearly 700 meetings where monitored. Some users had fewer than 20 meetings, while others had over 250. Most users had about 50% of their meetings delayed (this includes regularly scheduled meetings that were cancelled, for instance due to travel). Figure 12 shows that usually 50% or more of delayed meetings were autonomously delayed. In this graph, repeated delays of a single meeting are counted only once. The graphs show that the agents are acting autonomously in a large number of instances, but, equally importantly, humans are also often intervening, indicating the critical importance of *adjustable* autonomy in Friday agents.

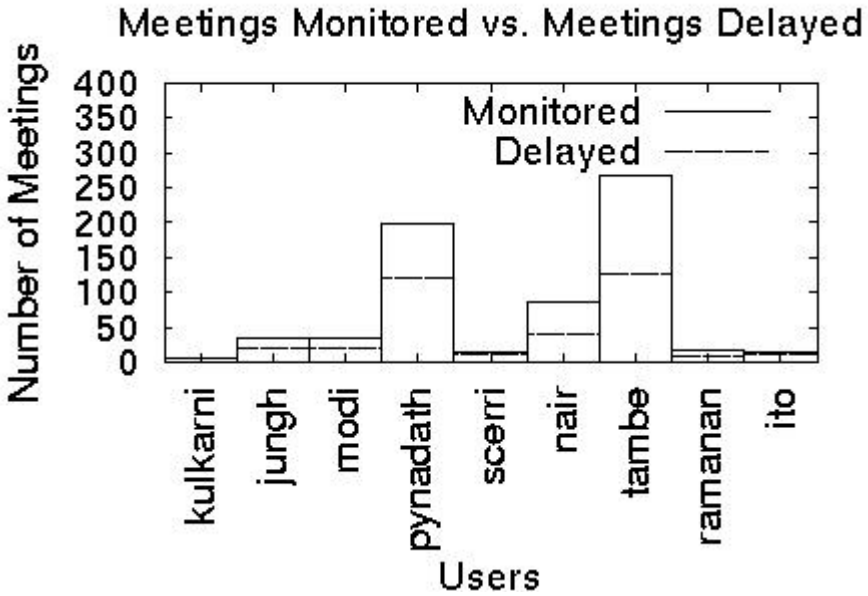


Figure 11: **Monitored vs. delayed meetings per user.**

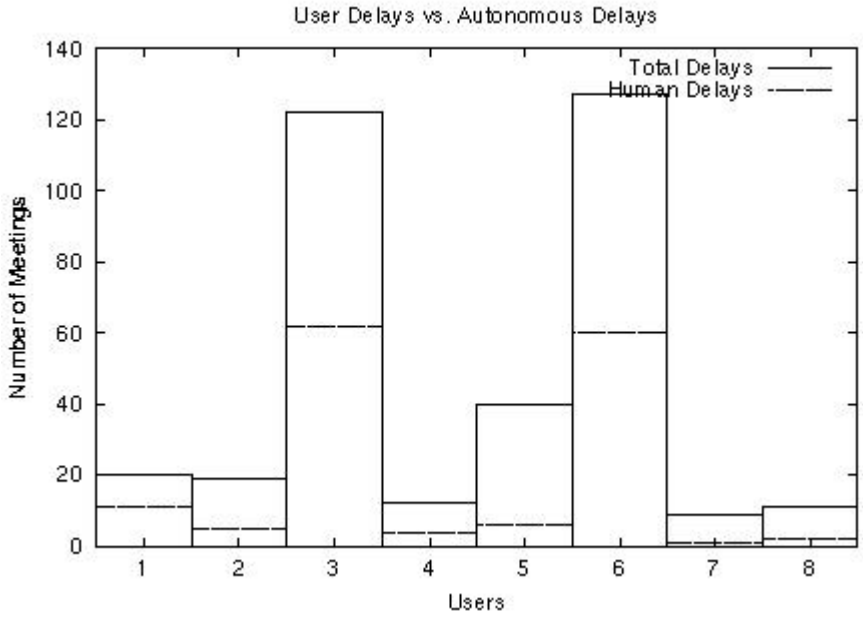


Figure 12: **Meetings delayed autonomously vs. by hand.**

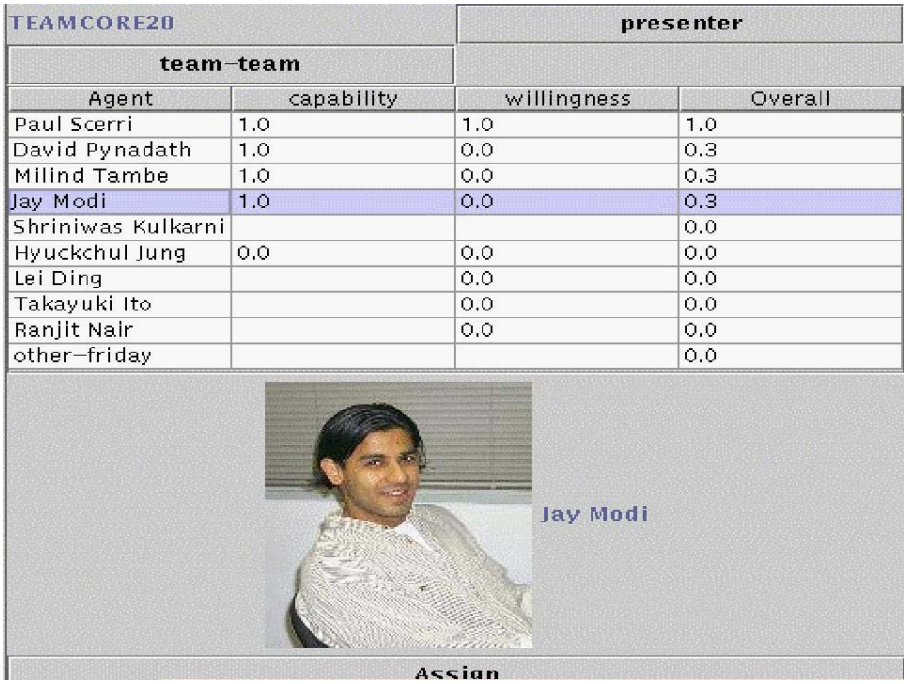


Figure 13: Sample screen shot from auction, used by Electric Elves to assign the presenter at a research group meeting.

For the seven-month period, the presenter for USC/ISI's Teamcore research group presentations was decided using auctions. Table 1 shows a summary of the auction results. Column 1 ("Date") shows the dates of the research presentations. Column 2 ("No. of Bids") shows the total number of bids received before a decision. A key feature is that auction decisions were made without all 9 users entering bids; in fact, in one case, only 4 bids were received. Column 3 ("Best bid") shows the winning bid. A winner typically bid <1,1>, i.e., indicating that the user it represents is both capable and willing to do the presentation — a high-quality bid. Interestingly, the winner on July 27 made a bid of <0,1>, i.e., not capable but willing. The team was able to settle on a winner despite the bid not being the highest possible, illustrating its flexibility. Finally, columns 4 ("Winner") and 5 ("Method") show the auction outcome. An 'H' in column 5 indicates that the auction was decided by a human, an 'A' indicates it was decided autonomously. In five of the seven auctions, a user was automatically selected to be presenter. The two manual assignments were due to exceptional circumstances in the group (e.g., a first-time visitor), again illustrating the need for AA.

Date	No. of bids	Best bid	Winner	Method
Jul 6	7	1,1	Scerri	H
Jul 20	9	1,1	Scerri	A
Jul 27	7	0,1	Kulkarni	A
Aug 3	8	1,1	Nair	A
Aug 31	4	1,1	Tambe	A
Sept 19	6	-,-	Visitor	H
Oct 31	7	1,1	Tambe	A

Table 1: Results for auctioning research presentation slot.

3.4 Travel Elves

In the past, although the proxies worked together as a team, each individual proxy was itself a monolithic system with a fixed set of capabilities. For greater flexibility (in terms of modification, technology transfer, etc.), we transformed each monolithic proxy into a *multiagent team* for the Travel Elves deployment. Therefore, each user now has a team of lightweight agents serving his/her needs. The composition of this team is now individualized to the user's particular environment and needs.

More specifically, Figure 14 illustrates the new proxy agent architecture, as deployed in the Travel Elves system. The focal point of this proxy architecture is a Grid-based proxy agent. This agent provides a point of contact for other Grid agents (e.g., the flight monitoring agent) that may wish to notify or query the user. It maintains a record of any information specific to its individual user (e.g., full name, preferred devices). It also maintains a record of all of the other agents that are part of the proxy team. As agents join and leave the proxy team, this Grid-based agent updates the proxy team's registration and capability description within the Grid registry. Thus, all agents can look up a user's proxy agent in the Grid registry to find out what interaction is possible with the particular user.

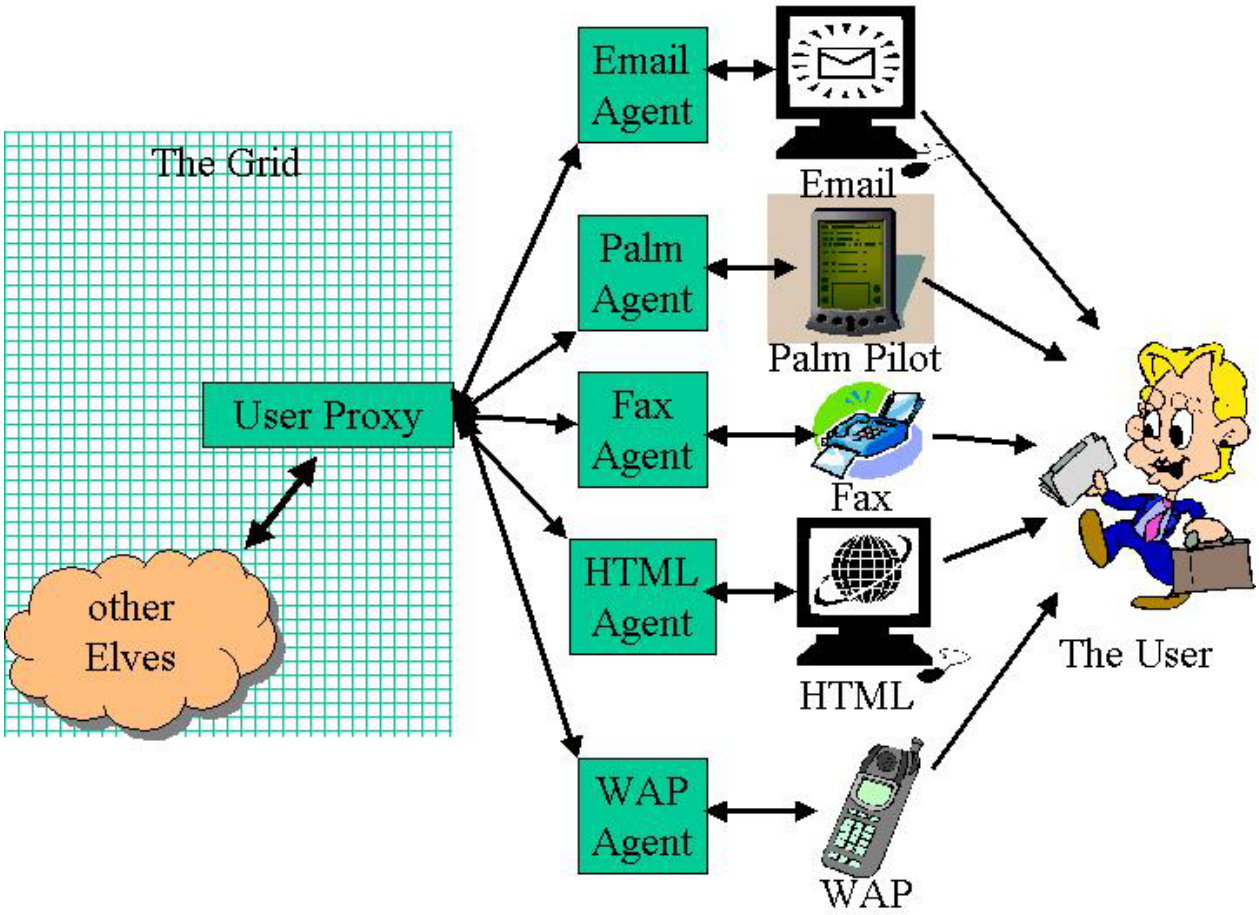


Figure 14: Agent architecture for user proxy subteam as deployed in Travel Elves.

In this deployment, the members of a user's proxy team consist of individual *device agents* to handle interactions with the user over various media. For example, if the user has a PDA, then the proxy team includes a specialized agent for interacting with this PDA. Likewise, a user with a SMS- or WAP-enabled cellular phone has a specialized agent in the proxy team for interacting through the device. A user can dynamically create his/her communication team through a Web page, at which point, the appropriate device agents begin operation and coordination amongst themselves.

In constructing these agents, we built on code that was part of the previously more ISI-specific incarnation of our user proxies. However, we extended the functionality by extracting the code (written in Python) and housing it in separate agents. Each agent can communicate with the Grid proxy agent using the KQML agent communication language. In addition, we generalized the operation of each agent so as to handle users who may be outside the ISI environment. Finally, we created special *device manager* agents that supported the dynamic creation of the specialized device agents. The presence of these device manager agents allow users to seamlessly register and unregister their devices through a Web page, while automatically maintaining the necessary control over the pool of operating agents.

3.4.1 Email Agents

The most popular medium for user notification is email. Most users have an email account that they check regularly. The email agents take any incoming messages for the user (in the form of a KQML message from the Grid-based proxy agent) and generate an appropriate email message to be sent through the mail server. The user can also specialize the email agent with respect to the size of messages sent. For users who will be viewing their email on a PC or workstation monitor, their email agents will provide very detailed messages. However, for users who wish to receive these emails on smaller, mobile devices (e.g., through SMS text messaging on a cellular phone), then their email agents will provide abbreviated content, so that the messages can still convey the necessary information within the restrictions of the display's size.

3.4.2 Palm Pilot Agents

These agents allow users to interact with their agent proxies through mobile Palm Pilot devices with Internet access. Unlike email, the Palm Pilot easily supports two-way communication with the user. We modified our original Palm OS-based client application to support users outside of ISI. The result is a mobile application that serves messages to the user through HTML pages that are readable on even the small display of the Palm Pilot. These pages provide the user with a history of current messages, as well as links to the contents of each message itself.

Our Palm Pilot client application also provides direct access to some of the Travel Elves functionality. For instance, a GPS device is connected to the Palm Pilot, a user can use the client application to generate a Grid query to the Ariadne restaurant finder agent with his/her specific latitude/longitude coordinates. Thus, our Palm Pilot agent provides ubiquitous access to the Electric Elves' functionality.

3.4.3 Fax Agents

These agents allow the agent proxies to communicate with their users through fax devices. In other words, if the user has an available fax number, this device agent can translate Grid messages into fax content and then send the fax to the user's number. The user does not have to perform any special configuration of the fax machine to receive these faxes; the device agent performs all of the necessary translation to format the original Grid message and make it readable.

3.4.4 HTML Agents

These agents allow users to interact with their agent proxies through a standard Web browser. The operation is roughly the same as that of the Palm agent, except that the user can typically interact with this agent through a PC or workstation. Thus, there is much more display room than with a Palm Pilot, allowing for more detailed messages. Furthermore, the user can interact with

this HTML agent through any Web browser on any machine anywhere.

3.4.5 WAP Agents

These agents allow users to interact with their agent proxies through a standard WML browser. The operation is roughly the same as that of the Palm and HTML agents, except that user typically interacts with this agent through a WAP-enabled cellular phone.

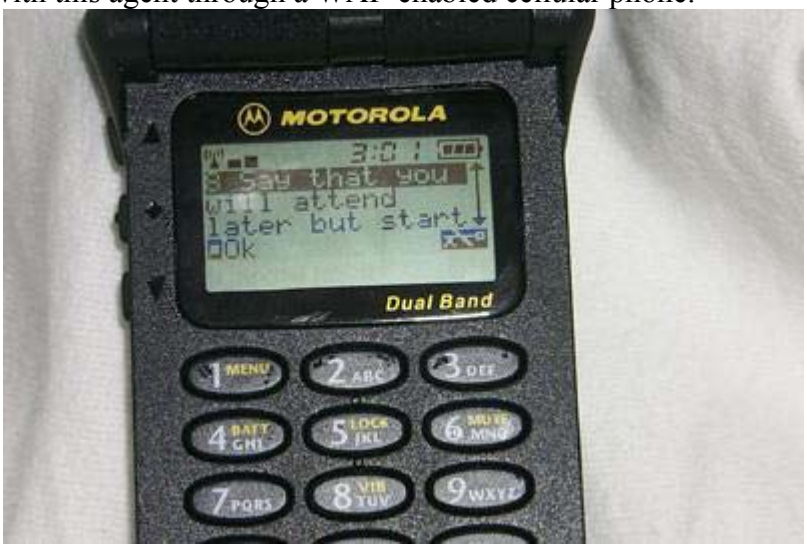


Figure 15: WAP-enabled cellular phone for wireless communication between proxy and user.

3.4.6 Speech Agents

These agents use text-to-speech software to create an auditory channel for informing the users. The channel is necessarily one-way, in that the message is spoken out loud to inform the user, but no reply is expected in return. This agent provides a "push" medium, in that the agent can transmit the message to the user immediately, without having to wait for the user to "pull" the information (e.g., by checking her/his email).

3.4.7 Display Agents

These agents use dialog boxes on the user's workstation display to transmit messages. The code for this display agent uses platform-independent code written in Python using the Tk widget set; thus, the user can run such agents on Windows, Unix, and Macintosh machines. Like the speech agents, these display agents are more of a "push" medium, in that the agent can pop up dialog boxes that will inform the user, without her/his having to actively ask for the information.

3.4.8 Fax Server Agent

The faxing device agents (described in Section 3.4.3) allow the Electric Elves agents to send faxes *to* the user. In many cases, it is desirable for these agents to send faxes *from* a user to another entity. For instance, in the flight monitoring task, when observing that a user's flight has been delayed late into the night, it is desirable to notify the user's hotel of the delay, so that the hotel holds the reservation until arrival. The agents automate this notification by making use of a *fax server agent*, capable of generating outgoing faxes to arbitrary people (e.g., the hotel in this example).

3.5 Travel Elves Deployment

As already mentioned, we have deployed the agents described in this report for general use, and for particular use by program manager, Dylan Schmorrow. Users can go to a registration Web page at `http://www.isi.edu/teamcore/elves`. Upon registering, a new Grid proxy is dynamically generated to handle all future interactions with the user. To do so, we run a Grid agent that listens for such registration requests. Upon receiving a request, it then starts up a new Grid proxy agent. This new agent then handles any future interactions with the user.

As part of this deployment, the program manager, Dylan Schmorrow, successfully registered through our Web page and now has an agent proxy team operating on his behalf. He registered an email device, so the team includes an email agent that will forward any notifications to the registered email account (which he can view on a mobile, Blackberry device). We have also provided him with a Palm Pilot, complete with our E-Elves Palm client application, and have started a corresponding Palm Pilot device agent.

The notifications sent to the user's proxy agent team come from other agents within the deployed Travel Elves system. These other agents provide various flight monitoring services (e.g., observing a change in schedule, air fare, etc.). These agents are also now deployed, and operation is ongoing.

As of this writing, there are 14 registered users, each with a Grid-based user proxy, and a total of 20 device agents. We have recently announced the availability of E-Elves to the general population in the ISI Intelligent Systems Division, so we expect this number to grow (i.e., as we gain users outside the E-Elves researchers and CoABS managers themselves). To ease the administrative burden of maintaining this agent population, we have developed a Web-based "control panel" that provides an easy mechanism for monitoring and repairing all of the agents in our E-Elves team. Figure 14 shows the interface, with the left-hand side panel listing the various functions supported.

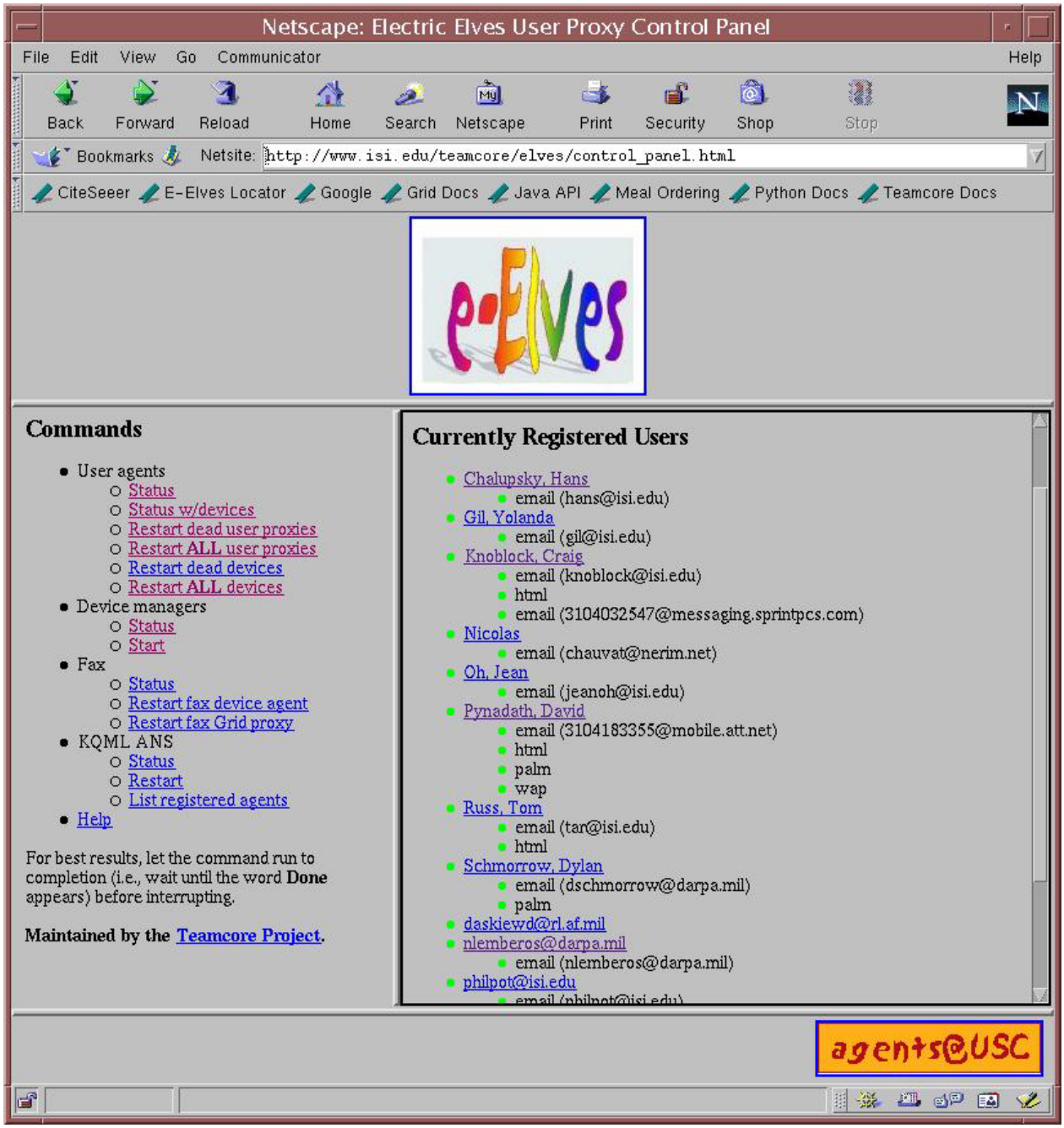


Figure 16: Web-based interface for administration of Electric Elves user agents and device teams.

4. Adjustable Autonomy

A central problem in adjustable autonomy is to determine whether and when transfers of

decision-making control should occur. A key challenge lies in the balance between two potentially conflicting goals. On one hand, to ensure that the highest-quality decisions are made, an agent can transfer control to a human user (or another agent) whenever that user has superior decision-making expertise. On the other hand, interrupting a user has high costs and the user may be unable to make and communicate a decision, so such transfers-of-control should be minimized. Previous work has examined several different techniques that attempt to balance these two conflicting goals and to thus address the transfer-of-control problem.

Our initial attempt at adjustable autonomy was inspired by Mitchell's CAP, an agent system for advising a user on scheduling meetings. As with CAP, each Friday tried to learn its user preferences using decision trees under Quinlan's C4.5. One problem became apparent when applying this technique in Electric Elves: a user would not grant autonomy to Friday in making certain decisions, but s/he would sometimes be unavailable to provide any input at decision time. Thus, a Friday could end up waiting indefinitely for user input and miscoordinate with its teammates. We therefore modified the system so that if a user did not respond within a fixed time limit, Friday acted autonomously based on its learned decision tree. Unfortunately, when we deployed the system in our research group, it led to some dramatic failures. For instance, one user's proxy erroneously volunteered him to give a presentation. C4.5 had overgeneralized from a few examples to create an incorrect rule. Although Friday tried asking the user at first, because of the timeout, it had to eventually follow the incorrect rule and take the undesirable autonomous action.

It was clear, based on this experience that the team context in Electric Elves would cause difficulties for existing adjustable-autonomy techniques that focused on solely individual human-agent interactions. When applied to interacting teams of agents and humans, where interaction between an agent and a human affects the interaction with other entities, these previous techniques can lead to dramatic failures. In particular, the presence of other entities as team members introduces an additional goal of maintaining coordination, which these previous techniques fail to address. Failures occur for two reasons. First, these previous techniques ignore team-related factors, such as costs to the team due to delays in decisions during such transfers of control. Second (and more important), these techniques use one-shot transfers of control, rigidly committing to one of two choices: (i) transfer control and wait for human input (choice *H*) or (ii) act autonomously (choice *A*). However, given interacting teams of agents and humans, both choices can lead to costly failures if the entity with control fails to make or report a decision in a way that maintains coordination. On the other hand, forcing a less capable entity to make a decision simply to avoid miscoordination can lead to poor decisions with significant consequences.

4.1 Transfer-of-control Strategies

To address the shortcomings of previous adjustable autonomy work in such domains, we developed a model of *transfer-of-control strategies*. A transfer-of-control strategy consists of a planned, conditional sequence of two types of actions: (i) actions to transfer decision-making control (e.g., from an agent to a user or vice versa); (ii) actions to change an agent's pre-specified coordination constraints with team members, rearranging activities as needed (e.g., reordering tasks to buy time to make the decision). The agent executes such a strategy by performing the actions in order, transferring control to the specified entity and changing coordination as

required, until some point in time when the entity currently in control exercises that control and makes the decision. Thus, the previous choices of *H* or *A* are just two of many different and possibly more complex transfer-of-control strategies. For instance, an *A*$\mathcal{D}$*AH* strategy implies that an agent *A*, initially attempts to make an autonomous decision. If the agent, *A*, makes the decision, the strategy execution ends there. However, there is a chance that it is unable to make the decision in a timely manner, perhaps because its computational resources are busy with higher priority tasks. To avoid miscoordination, the agent executes a $\mathcal{D}$ action that changes the coordination constraints on the activity. For example, a $\mathcal{D}$ action could inform other agents that the coordinated action will be delayed, thus incurring a cost of inconvenience to others, but also buying more time to make the decision. If it still cannot make the decision, it will eventually take action *H*, transferring decision-making control to the user and waiting for a response. In general, strategies can involve all available entities and contain many actions to change coordination constraints. While such strategies may be useful in single-agent single-human settings, they are particularly critical in general multiagent settings.

We have developed a decision-theoretic model of such strategies, that allows the expected utility of a strategy to be calculated and, hence, strategies to be compared. A key adjustable autonomy problem lies in the selection of the best possible transfer-of-control strategy, i.e., one that provides the benefit of high-quality decisions without risking significant costs in interrupting the user and miscoordination with the team. Furthermore, an agent must select the right strategy despite significant uncertainty.

Markov decision processes (MDPs) are a natural choice for implementing such reasoning, because they explicitly represent costs, benefits and uncertainty as well as doing lookahead to examine the potential consequences of sequences of actions. We conducted detailed experiments on MDP-based adjustable-autonomy reasoning used in E-Elves over the course of several months. For instance, as we varied the relative importance of central factors (e.g., cost of miscoordination), the resulting MDP policies varied in a desirable way (e.g., the agent made more decisions autonomously if the cost of transferring control to other entities increased). Other experiments reveal a phenomenon not reported before in the literature: an agent may act more autonomously when coordination change costs are either too low or too high, but in a 'middle' range, the agent tends to act less autonomously. Despite the unpredictability of the user's behavior and the agent's limited sensing abilities, the MDPs in use within E-Elves consistently made sensible adjustable autonomy decisions. Moreover, many times the agent performed several transfers of control to cope with contingencies such as a user not responding. One lesson learned when actually deploying the system was that sometimes users wished to influence the adjustable autonomy reasoning, e.g., to ensure that control was transferred to them in particular circumstances. To allow users to influence the adjustable autonomy reasoning, we introduced safety constraints that allow users to prevent agents from taking particular actions or ensuring that they do take particular actions. These safety constraints provide guarantees on the behavior of the adjustable autonomy reasoning, making the basic approach more generally applicable and, in particular, making it more applicable to domains where mistakes have serious consequences.

4.2 Evaluation of AA Strategies

The most emphatic evidence of the success of the MDP approach is that, since replacing the C4.5 implementation, the agents have *never* repeated any of the catastrophic mistakes made under that

earlier implementation. For example, Friday avoids an earlier C4.5-based error where it selected several small, 5-minute delays instead of a single longer delay. Under the MDP approach, Friday instead selects a strategy with a single, large $\mathcal{D}$ action, because it has a higher EU than a strategy with many small $\mathcal{D}$s (e.g., $\mathcal{DDDD}$). Friday avoids an erroneous cancellation (which were observed under C4.5), because the large cost associated with an erroneous cancel action significantly penalizes the EU of a cancellation. Friday instead chooses the higher-EU strategy that first transfers control to a user before taking such an action autonomously. Friday avoids other errors observed under our initial implementation by selecting strategies in a situation-sensitive manner. For instance, if the agent's decision-making quality is low (i.e., high risk), then the agent can perform a coordination-change action to allow more time for user response or for the agent itself to get more information. This indicates that a reasonably appropriate strategy was chosen in each situation. Although the current agents do occasionally make mistakes, these errors are typically on the order of transferring control to the user a few minutes earlier than may be necessary. Thus, the agents' decisions have been reasonable, though not always optimal. The inherent subjectivity in user feedback makes a determination of optimality difficult.

5. Monitoring Agent Teams

Recent years have seen tremendous growth of applications involving distributed multi-agent teams, formed of agents that collaborate on a specific joint task, including the two domains of Teamcore's application described here. This growth has led to increasing need for monitoring techniques that allow a synthetic agent or human operator to monitor and identify the state of the distributed team. Researchers have already identified the critical role of monitoring in visualization, in identifying failures in execution, in providing advice to improve performance, and in facilitating collaboration between the monitoring agent and the members of the team.

Motivated by the difficulties in monitoring the distributed agents in TIE 1 and in Electric Elves, we focused on monitoring cooperative agent teams by overhearing their internal communications. This allows a human operator or a synthetic agent to monitor the coordinated execution of a task, by listening to the messages team-members exchange with each other. It contrasts with previous techniques that are impractical in settings where direct observations of the team members are unavailable (e.g., when an observer at ISI in California wishes to monitor agents running TIE 1 at Global Infotek in D.C.), or in large-scale applications composed of *already-deployed* agents that are dynamically integrated to jointly execute a task.

For example, one common technique, *report-based monitoring,* requires each monitored team-member to communicate its state to the monitoring agent at regular intervals, or at least whenever the team-member changes its state. Such reporting provides the monitoring agent with accurate information on the state of the team. Unfortunately, report-based monitoring suffers from several difficulties in monitoring large deployed teams of interest in the real-world: First, it requires intrusive modifications to the behavior of agents, such that they report their state as needed by the different monitoring applications. However, since agents are already deployed, such repeated modifications to the behavior of the agents are difficult to implement and complex to manage. In particular, legacy and proprietary systems are notoriously expensive to modify (e.g., the notorious modifications to address Y2K bugs). Second, the bandwidth requirements of report-based monitoring (which relies on communication channels) can be unrealistic, as other

researchers have repeatedly pointed out. In addition, network delays and unreliable or lossy communication channels are a key concern with report-based monitoring approaches.

We therefore advocate an alternative monitoring approach, based on multi-agent keyhole plan-recognition, which has already had limited success in monitoring small-scale multiagent systems. In this approach, the monitoring system (deployed by the human operator of the team) infers the unobservable state of the agents based on their observable actions, using knowledge of the plans that give rise to the actions. This approach is non-intrusive, requiring no changes to agents' behaviors; and it allows for changes in the requested monitoring information. It assumes access to knowledge of plans that may explain observable actions—however this knowledge is readily available to the monitoring system as it is deployed by the human operator of the team. Furthermore, a plan-recognition approach can rely on inference to compensate for occasional communication losses, and can therefore be robust to communication failures.

In general, the only observable actions of agents in a distributed team are their *routine* communications, which the agents exchange as part of task execution. Fortunately, the growing popularity of general-purpose agent-integration tools and increased standardization of aspects of agent communications provide increasing opportunities for observing and interpreting inter-agent communications. We assume that monitored agents are truthful in their messages, since they are communicating to their teammates; and that they are not attempting to deceive the monitoring agent or prevent it from overhearing (as it is deployed by the human operator of the team). Given a (possibly stochastic) model of the plans that the agents may be executing, a monitoring system using plan-recognition can infer the current state of the agents from such observed routine messages.

However, the application of plan-recognition techniques for overhearing poses significant challenges. First, a key characteristic of the overhearing task is the scarcity of observations. Explanations for overheard messages (i.e., the observed actions) can sometimes be fairly easy to disambiguate, but uncertainty arises because there are relatively few of them to observe: team members cannot and do not in practice continuously communicate among themselves about their state. Thus team-members change their state while keeping quiet. Another key characteristic of overhearing is that the observable actions are inherently *multi-agent actions*: When agents communicate, it is only a single agent that *sends* the messages. The others implicitly act their role in the communications by *listening*. Yet despite the scarcity of observable communications, and the multi-agent nature of the observed actions, a monitoring system must infer the state of all agents in the team, at all times. Previous investigations of multi-agent plan-recognition have typically made the assumption that all changes to the state of agents have an observable effect: Uncertainty resulted from ambiguity in the explanations for the observed actions. Furthermore, these investigations have addressed settings where observable actions were individual: Each agent was observed while acting.

In addition to these challenges that are unique to overhearing, a monitoring system must address additional challenges stemming from the use of monitoring in service of visualization. The representation and algorithms must support soft real-time response; reasoning must be done quickly to be useful for visualization. Furthermore, real-world applications demand techniques that can scale up as the number of agents increases, for monitoring large teams. However, many current representations for plan-recognition are computationally intense (e.g., dynamic Bayesian networks), or only address single-agent recognition tasks. Multi-agent plan-recognition investigations have typically not explicitly addressed scalability concerns.

To address these novel issues, we developed and implemented OVERSEER, which is capable of monitoring large distributed applications composed of previously-deployed agents. OVERSEER

builds on previous work in multi-agent plan-recognition by utilizing knowledge of the relationships between agents to understand how their decisions interact. However, as previous techniques proved insufficient, OVERSEER includes a number of novel multi-agent plan-recognition techniques that address the scarcity of observations, as well as the severe response-time and scale-up requirements imposed by realistic applications. Key contributions include: (i) a *linear time* probabilistic plan-recognition representation and associated algorithms, which exploit the nature of observed communications for efficiency; (ii) a method for addressing unavailable observations by exploiting knowledge of the *social procedures* of teams to effectively predict (and hence effectively monitor) future observations during normal and failed execution, thus allowing inference from lack of such observations; and (iii) YOYO*, an algorithm that uses knowledge of the team organizational structure (*team-hierarchy*) to model the agent team (with all the different parallel activities taken by individual agents) using a single structure, instead of modeling each agent individually. YOYO* sacrifices some expressivity (the ability to accurately monitor the team in certain coordination failure states) for significant gains in efficiency and scalability.

We performed a rigorous evaluation of OVERSEER's different monitoring techniques using the CoABS TIE 1 as a testbed domain and showed that our techniques result in significant boosts to OVERSEER's monitoring accuracy and efficiency, beyond techniques explored in previous work. We evaluated OVERSEER's capability to address lossy observations, a key concern with report-based monitoring. Furthermore, we evaluated OVERSEER's performance in comparison with human expert and novice monitors, and showed that OVERSEER's performance is comparable to that of human experts, despite the difficulty of the task, and OVERSEER's reliance on computationally-simple techniques. One of the key lessons that we draw in OVERSEER is that a combination of computationally-cheap multi-agent plan-recognition techniques, exploiting knowledge of the expected structures and interactions among team-members, can be competitive with approaches which focus on accurate modeling of individual agents (and may be computationally expensive).

6. Teamwork Theory

Our work on determining optimal strategies for adjustable autonomy led us to further examine our teamwork-based integration architecture. Research in teamwork theory has built the foundations for successful practical agent team implementations, including our own Teamcore architecture. On the forefront are theories based on belief-desire-intentions (BDI) frameworks (e.g., joint intentions, SharedPlans, and others) that have provided prescriptions for coordination in practical systems. These theories have inspired the construction of practical, domain-independent teamwork models and architectures (including STEAM and Teamcore), successfully applied in a range of complex domains.

Yet, two key shortcomings limit the scalability of these BDI-based theories and implementations. First, there are no techniques for the quantitative evaluation of the degree of *optimality* of their coordination behavior. While optimal coordination may be impractical in real-world domains, such analysis would aid us in comparison of different theories/models and in identifying feasible improvements. One key reason for the difficulty in quantitative evaluation of most existing teamwork theories is that they ignore the various uncertainties and costs in real-

world environments. For instance, joint intentions theory prescribes that team members attain mutual beliefs in key circumstances, but it ignores the cost of attaining mutual belief (e.g., via communication). Implementations that blindly follow such prescriptions could engage in highly suboptimal coordination. On the other hand, practical systems have addressed costs and uncertainties of real-world environments. For instance, STEAM extends joint intentions with decision-theoretic communication selectivity. Unfortunately, the very pragmatism of such approaches often necessarily leads to a lack of theoretical rigor, so it remains unanswered whether STEAM's selectivity is the best an agent can do, or whether it is even necessary at all. The second key shortcoming of existing teamwork research is the lack of a characterization of the computational complexity of various aspects of teamwork decisions. Understanding the computational advantages of a practical coordination prescription could potentially justify the use of that prescription as an approximation to optimality in particular domains.

To address these shortcomings, we propose a new complementary framework, the *COMmunicative Multiagent Team Decision Problem (COM-MTDP)*, inspired by work in *economic team theory*. As in that framework, our definition of a team assumes only a common goal (i.e., a joint utility function). Unlike typical teamwork frameworks, we make no other assumptions about the team's behavior (e.g., the teammates form a joint commitment, communicate to attain mutual belief, etc.). We view these more intermediate concepts as the *means* by which agents improve their overall performance, not ends in themselves. For example, while mutual belief has no inherent value, our COM-MTDP model can quantify the improved performance that we would expect from a team that attains mutual belief about important aspects of its execution. While our COM-MTDP model borrows from a theory developed in another field, we make several contributions in applying and extending the original theory, most notably adding explicit models of communication and system dynamics. With these extensions, the COM-MTDP generalizes other recently developed multiagent decision frameworks, such as decentralized POMDPs.

This paper demonstrates three new types of teamwork analyses made possible by the COM-MTDP model. First, we analyze the computational complexity of teamwork within subclasses of problem domains. For instance, some researchers have advocated teamwork without communication. We use the COM-MTDP model to show that, in general, the problem of constructing optimal teams without communication is NEXP-complete, but allowing free communication reduces the problem to be PSPACE-complete. This paper presents a breakdown of the complexity of optimal teamwork over problem domains classified along the dimensions of observability and communication cost.

Second, the COM-MTDP model provides a powerful tool for comparing the *optimality* of different coordination prescriptions across classes of domains. Indeed, we illustrate that we can encode existing team coordination strategies by within a COM-MTDP for evaluation. For our analysis, we selected two joint intentions-based approaches from the literature: one using the approach realized within GRATE* and the joint responsibility model of Jennings, and another based on STEAM. Through this encoding, we derive the conditions of optimality for these coordination strategies, and the complexity of the decision problems addressed by these coordination strategies. Furthermore, we also derive a novel coordination algorithm that outperforms these existing coordination strategies in optimality, though not in efficiency. The end result is a *well-grounded characterization of the complexity-optimality tradeoff* among various means of team coordination.

Third, we can use the COM-MTDP model to empirically analyze a specific domain of interest. We have implemented reusable, domain-independent algorithms that allow one to

evaluate the optimality of different prescriptive policies within a problem domain represented as a COM-MTDP. We apply these algorithms in an example domain to empirically evaluate the aforementioned coordination strategies, characterizing the optimality of each strategy as a function of the properties of the underlying domain. For instance, Jennings reports experimental results indicating that his joint responsibility teamwork model leads to lower waste of community effort than competing methods of accomplishing teamwork. With our COM-MTDP model, we were able to demonstrate the benefits of Jennings' approach under many configurations of our example domain. However, in precisely characterizing the types of domains that showed such benefits, we also identified domains where these competing methods may actually perform better. In addition, we can use our COM-MTDP model to re-create and explain previous work that noted an instance of suboptimality in a STEAM-based, real-world implementation. While this previous work treated that suboptimality as anomalous, our COM-MTDP re-evaluation of the domain demonstrated that the observed suboptimality was a symptom of STEAM's general propensity towards extraneous communication in a significant range of domain types. Both the algorithms and the example domain model are available for public use in an online appendix (`http://www.isi.edu/teamcore/COM-MTDP/`).

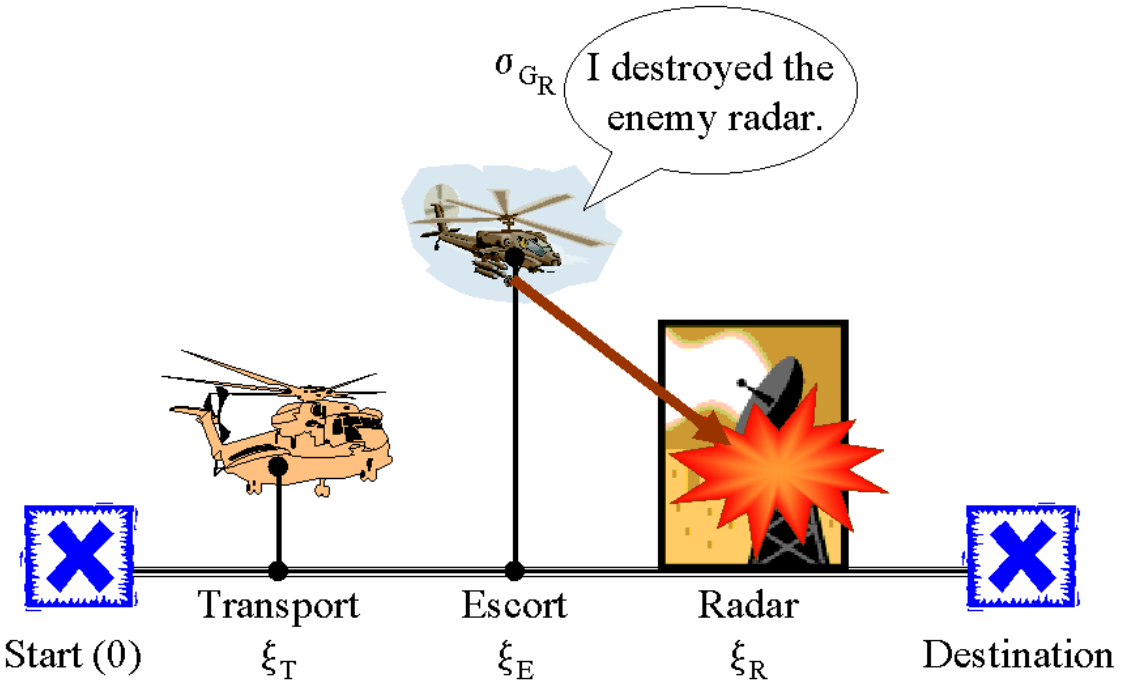


Figure 17: Helicopter scenario used as example domain for COM-MTDPs.

7. Technology Transfer

In addition to making the Travel Elves technology generally available, we are also looking into ways to transfer our COM-MTDP technology into other arenas.

7.1 Psychological Operations

In work being conducted on a subcontract through the Institute for Defense Analyses to the Psychological Operations (Psyops) branch of Special Forces, we are looking at using the COM-MTDP model to represent and reason about human societies. The goal is to use agents to represent various individuals or subpopulations (and the relationships among them). We then use the COM-MTDP communication policies to represent message campaigns. With such a mapping, we can use our general-purpose COM-MTDP policy evaluation algorithms to measure the potential effectiveness of candidate message campaigns and provide valuable feedback to human Psyop analysts.

7.2 Disaster Rescue

We are also looking into the use of the COM-MTDP framework for analyzing the coordination of rescue personnel responding to an urban disaster. We again use agents to represent the personnel (e.g., fire trucks, policemen, ambulances) and the policy evaluation algorithms to measure the effectiveness of candidate coordination strategies. We can also potentially develop new algorithms for *generating* the best possible coordination strategy for a given disaster rescue scenario.

8. Personnel

- Milind Tambe (PI)
- Wei-Min Shen (Co-PI)
- David V. Pynadath (Research Scientist)
- Paul Scerri (Research Scientist)

9. Publications

[1] Hans Chalupsky, Yolanda Gil, Craig A. Knoblock, Kristina Lerman, Jean Oh, David V. Pynadath, Thomas A. Russ, and Milind Tambe. Electric Elves: Applying agent technology to support human organizations. To appear in *Proceedings of the Innovative Applications of Artificial Intelligence Conference*, 2001.

[2] Gal Kaminka, David V. Pynadath, and Milind Tambe. Monitoring deployed agent teams. In *Proceedings of the International Conference on Autonomous Agents*, pages 308–315, 2001.

[3] Gal Kaminka, David V. Pynadath, and Milind Tambe. Monitoring teams by overhearing: A

multi-agent plan-recognition approach. *Journal of Artificial Intelligence Research*, page To appear, 2002.

[4] David V. Pynadath, Paul Scerri, and Milind Tambe. MDPs for adjustable autonomy in real-world multi-agent environments. In *Proceedings of the AAAI Spring Symposium on Game Theoretic and Decision Theoretic Agents*, pages pp. 107–116, 2001.

[5] David V. Pynadath and Milind Tambe. Electric Elves: Adjustable autonomy in real-world multi-agent environments. In Kerstin Dautenhahn, Alan Bond, Dolores Canamero, and Bruce Edmonds, editors, *Socially Intelligent Agents - Creating Relationships with Computers and Robots*. 2001.

[6] David V. Pynadath and Milind 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.

[7] David V. Pynadath and Milind Tambe. An automated teamwork infrastructure for heterogeneous software agents and humans. *Journal of Autonomous Agents and Multi-Agent Systems, Special Issue on Infrastructure and Requirements for Building Research Grade Multi-Agent Systems*, page to appear, 2002.

[8] David V. Pynadath and Milind Tambe. The communicative multiagent team decision problem: Analyzing teamwork theories and models. *Journal of Artificial Intelligence Research*, page to appear, 2002.

[9] David V. Pynadath and Milind Tambe. Multiagent teamwork: Analyzing the optimality and complexity of key theories and models. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi-Agent Systems*, page to appear, 2002.

[10] David V. Pynadath and Milind Tambe. Team coordination among distributed agents: Analyzing key teamwork theories and models. In *Proceedings of the AAAI Spring Symposium on Intelligent Distributed and Embedded Systems*, pages 57–62, 2002.

[11] David V. Pynadath, Milind Tambe, Yigal Arens, Hans Chalupsky, Yolanda Gil, Craig Knoblock, Haeyoung Lee, Kristina Lerman, Jean Oh, Surya Ramachandran, Paul S. Rosenbloom, and Thomas Russ. Electric Elves: Immersing an agent organization in a human organization. In *AAAI Fall Symposium on Socially Intelligent Agents: The Human in the Loop*, pages 150–154, 2000.

[12] David V. Pynadath, Milind Tambe, and Nicolas Chauvat. Rapid integration and coordination of heterogeneous, distributed agents for collaborative enterprises. In *Proceedings of the DARPA-JFACC Symposium on Advances in Enterprise Control*, pages 171–176, 1999.

[13] David V. Pynadath, Milind Tambe, Nicolas Chauvat, and Lawrence 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")*, pages 77–91, 1999.

[14] David V. Pynadath, Milind Tambe, Nicolas Chauvat, and Lawrence Cavedon. Toward team-oriented programming. In Nicholas R. Jennings and Yves Lespérance, editors, *Intelligent Agents VI: Agent Theories, Architectures and Languages*, pages 233–247. Springer-Verlag, 1999.

[15] Paul Scerri, David V. Pynadath, and Milind Tambe. Adjustable autonomy in real-world multi-agent environments. In *Proceedings of the Conference on Autonomous Agents*, pages 300–307, 2001.

[16] Paul Scerri, David V. Pynadath, and Milind Tambe. Why the elf acted autonomously: Towards a theory of adjustable autonomy. In *Proceedings of the International Joint*

Conference on Autonomous Agents and Multi-Agent Systems, page To appear, 2002.

[17] Milind Tambe and David V. Pynadath. Towards heterogeneous agent teams. In Vladimir Marik and Olga Stepankova, editors, *Multi-Agent Systems and Applications, ACAI-01 Proceedings*. 2001.

[18] Milind Tambe, David V. Pynadath, and Nicolas Chauvat. Building dynamic agent organizations in cyberspace. *IEEE Internet Computing*, 4(2), March/April 2000.

[19] Milind Tambe, David V. Pynadath, Nicolas Chauvat, Abhimanyu Das, and Gal A. Kaminka. Adaptive agent integration architectures for heterogeneous team members. In *Proceedings of the International Conference on MultiAgent Systems*, pages 301–308, 2000.

[20] Milind Tambe, Wei-Min Shen, Maja Mataric, David V. Pynadath, Dani Goldberg, Pragnesh Jay Modi, Zhun Qiu, and Behnam Salemi. Teamwork in cyberspace: Using TEAMCORE to make agents team-ready. In *Proceedings of the AAAI Spring Symposium on Agents in Cyberspace*, pages 136–141, 1999.