Adaptive Agent Integration Architectures for Heterogeneous Team Members

Milind Tambe, David V. Pynadath, Nicolas Chauvat
Abhimanyu Das, Gal A. Kaminka
Information Sciences Institute and Computer Science Department
University of Southern California
4676 Admiralty Way, Marina del Rey, CA 90292
{tambe,pynadath,nico,das,galk}@isi.edu

Abstract

With the proliferation of software agents and smart hardware devices there is a growing realization that large-scale problems can be addressed by integration of such standalone systems. This has led to an increasing interest in integration architectures that enable a heterogeneous variety of agents and humans to work together. These agents and humans differ in their capabilities, preferences, the level of autonomy they are willing to grant the integration architecture and their information requirements and performance. The challenge in coordinating such a diverse agent set is that potentially a large number of domain-specific and agent-specific coordination plans may be required. We present a novel two-tiered approach to address this coordination problem. We first provide the integration architecture with general purpose teamwork coordination capabilities, but then enable adaptation of such capabilities for the needs or requirements of specific individuals. A key novel aspect of this adaptation is that it takes place in the context of other heterogeneous team members. We are realizing this approach in an implemented distributed agent integration architecture called Teamcore. Experimental results from two different domains are presented.

1 Introduction

With the ever increasing number of information-gathering agents, user agents, agents in virtual environments, smart hardware devices and robotic agents, there is a growing interest in agent integration architectures. Such architectures enable a heterogeneous set of agents and humans to work together to address large-scale problems not solvable by any particular individual[3, 2, 5].

Such agent integration architecture must address several important issues, such as locating relevant agents (or humans) for a task, facilitating their collaboration and monitoring their performance. This paper focuses on the challenge of facilitating agent collaboration in the context of heterogeneous agents, which have different capabilities, developers, and preferences. For instance, humans may differ in their requirements for obtaining coordination information and the cost they are willing to pay to obtain such information. Humans may also differ in the types of coordination decisions they will allow (or want) automated in the agent integration architecture. Software agents have still differing requirements for information and automated coordination. Such heterogeneity leads to the difficulty of encoding large numbers of special purpose coordination plans, specialized not only for each new domain, but also tailored for each individual agent requirements. Furthermore, given that these requirements may vary over time, these plans would need to be modified frequently.

Our approach to addressing the above challenge is to first provide the integration architecture with general-purpose teamwork coordination capabilities, and then to adapt such capabilities (via machine learning) for the needs and performance of specific individuals. General teamwork knowledge avoids the need to write large numbers of coordination plans for each new domain and agent. Adaptation based on this foundation enables the integration architecture to cater to individual coordination needs and performance. The foundation of the teamwork knowledge is critical for adaptation here, since learning all of the coordination knowledge from scratch for each case would be very expensive.

We are building a distributed agent integration architecture called Teamcore. Here, the agents or humans to be coordinated are each assigned Teamcore proxies, where the proxies work as a team. Each proxy contains Steam[11, 12], a general teamwork model that automates its coordination with other proxies in its team. Starting with this teamwork model, the proxies currently adapt via four different methods to the agents they represent, where each method covers a different aspect of a proxy's participation in the team activity. More specifically: (1) A proxy's *adaptive autonomy* refers to its adapting its level of decision-making autonomy in its team activities, so that it defers some/many decisions to the human or agent it represents; (2) The proxies' *adaptive execution* refers to their collectively adapting their plans at execution-time in response to an agent's varying performance; (3) The proxies' *adaptive monitoring* refers to their collectively adapting their communication patterns in response to an agent's differing requirements for execution monitoring; (4) A proxy's *adaptive information delivery* refers to its adapting to an agent's communication costs and reliabilities when delivering relevant coordination information. A key novelty in our approach is that adaptation is done in the context of a team, not necessarily just an individual proxy. For instance, the proxies may cause the team to communicate more to improve monitoring.

We begin this paper by presenting the Teamcore architecture, and its application in two complex domains. These applications motivate the need for Teamcore's adaptation, which is discussed next.

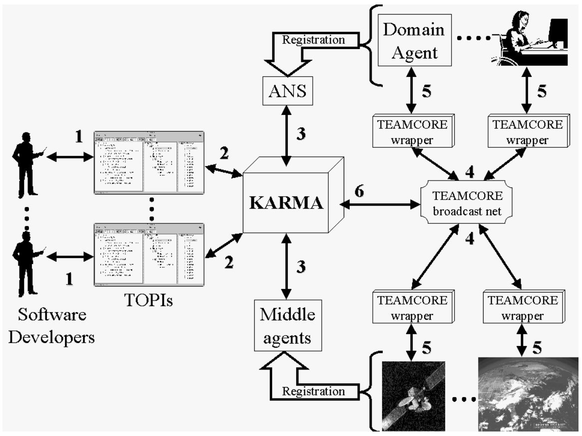

Figure 1. The overall Karma-TEAMCORE framework.

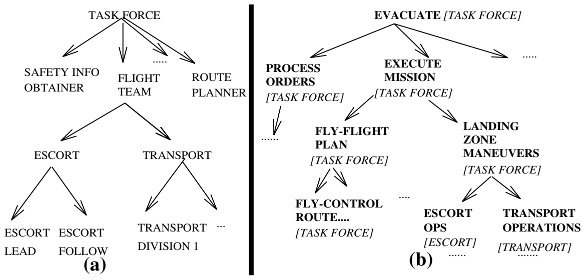

Figure 2. A team-oriented program.

2 Teamcore Framework

Figure 1 shows the overall Teamcore agent integration framework. The numbered arrows show the stages of interactions in this system. In stage 1, human developers interact with TOPI (team-oriented programming interface) to specify a team-oriented program, consisting of an organization hierarchy and hierarchical team plans. As an example, Figure 2 shows an abbreviated team-oriented program for the evacuation domain. Figure 2-a shows the organization hierarchy and Figure 2-b shows the plan hierarchy. Here, high-level team plans, such as **Evacuate**, typically decompose into other team plans and, ultimately, into leaf-level plans, that are executed by individuals. There are teams assigned to execute the plans, e.g., *Task Force* team is assigned to jointly execute **Evacuate**, while *Escort* subteam is assigned to the **Escort-operations** plan. These teams or individual roles are as yet not matched with actual agents.

TOPI in turn communicates the team-oriented program to Karma (stage 2). Karma is an *agent resources manager* — it queries (stage 3) different middle agents and ANS services for the "domain agents" (which may include diverse software agents or humans) with expertise relevant to the team-oriented program specified in stage 1. Located domain agents are matched to specific roles in the team plans (by Karma or developer or both). In stage 4, the Teamcore proxies jointly execute the team-oriented program. Here, each domain agent is assigned a Teamcore proxy. The proxies work as a team in executing the team plans, autonomously coordinating among themselves by broadcasting information via multiple broadcast nets (stage 4). Teamcores also communicate with the domain agents (stage 5). Karma monitors and records information about agent performance (stage 6). All communications occur in KQML.

A key feature of our framework is the proxies' in-built Steam domain-independent teamwork model. Steam provides a Teamcore with three sets of domain-independent teamwork reasoning rules: (i) *Coherence preserving* rules require team members to communicate with others for coherent initiation and termination of team plans; (ii) *Monitor and repair* rules ensure that team members substitute for other critical team members who may have failed in their roles; (iii) *Selectivity-in-communication* rules use decision theory to weigh communication costs and benefits to avoid excessive communication. Armed with these rules, the proxies automatically execute much of the required coordination, without it being explicitly included in the team oriented program. For instance, if a *domain agent* in *Task Force* executing **Evacuate** in Fig 1 were to fail, Teamcore proxies will automatically ensure that another team member (domain agent) with similar capabilities will substitute in the relevant role — such coordination is not explicitly programmed in the team-oriented program. Please see [13] for more details on the Teamcore framework.

2.1 Application 1: Evacuation Rehearsal

We have applied the Teamcore framework to the problem of rehearsing the evacuation of civilians from a threatened location. Here, an integrated system must enable a human commander (the user) 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 itself plan routes to avoid known obstacles, dynamically obtain information about enemy threats, and change routes when needed. The software developer was able to create a team-oriented program for this problem, using the following agents:

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) 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]

As seen above, these agents have different developers, they are written in different languages for different operating systems, they may be distributed geographically and 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).

We successfully used the Teamcore framework to build and execute a team-oriented program for evacuation mission rehearsal from these agents. An abbreviated portion of the program is seen in Fig 2. This program has about 40 team plans. There are 11 Teamcore proxies for the 11 agents, which execute this program by automatically communicating with each other (exchanging about 100 messages), while also correctly communicating with the domain agents.

2.2 Application 2: Assisting Human Collaboration

We are also using Teamcore to build an application to assist human teams in routine coordination activities in industrial or research organizations, using our own research team as a testbed. In this application, each human user has a Teamcore proxy that coordinates with other proxies on behalf of its user. These proxies communicate with the users using their workstation screens or their hand-held wireless personal digital assistants (PDAs). The distributed Teamcore architecture is well-suited in this domain, since each human maintains control on its own Teamcore and its information, rather than centralizing it.

Our current focus is facilitating coordination of meetings within our team or with visitors, at our institute or outside. For instance if currently an individual gets delayed (e.g., because she is finishing up results), other meeting attendees end up wasting time waiting or attempting to reach those missing. To help avoid such miscoordination, a Teamcore proxy keeps track of its user's scheduled meetings (by monitoring his/her calendar). These meetings are essentially the team plans to be executed jointly by the different Teamcores. Using Steam rules, the Teamcore proxies ensure coherent beliefs about the current state of the meeting. In particular, the proxies track the user's whereabouts (e.g., by using idle time on the user's workstations), and automatically inform other meeting attendees about meeting delays or about absentees. The proxies also automatically communicate with user's PDAs. Additionally, if an absent team member was playing an important role at the meeting, such as leading a discussion, Teamcore proxies attempt to get another person with similar capabilities to take over.

3 Adapting to team member heterogeneity

While the promising results of the applications discussed above indicate the benefits of founding the integration architecture on a proven model of teamwork, they also indicate ways in which the architecture must adapt to agent heterogeneity. The following subsections present four different methods of adaptation, each using a suitable technique. The overall theme in these adaptations is that in interacting with a heterogeneous team member (who may be human), the Teamcore proxies either adapt together as a team, or a single proxy adapts in the context of the team.

3.1 Adapting the level of autonomy

A key challenge in integrating heterogeneous agents is that they may have differing requirements with respect to the autonomy of the integration architecture to make decisions on their behalf. For instance, in the human collaboration application discussed in Section 2.2, a Teamcore proxy may commit its human user to substitute for a missing discussion leader, knowing that its user is proficient in the discussion topics. However, the human may or may not want the proxy to autonomously make this commitment. The decision may vary from person to person, and may depend on many diverse factors. Conversely, though, restricting the proxy to always confirm its decision with the user is also undesirable, since it would then often overwhelm the user with confirmation requests for trivial decisions.

Thus, it is important that a proxy have the right level of autonomy. Yet, to avoid hand-tuning such autonomy for each human (or agent), it is critical for a proxy to automatically adapt its autonomy to a suitable level. We rely on a supervised learning approach based on user feedback. Here, a key issue that contrasts our work with previous work on autonomy adaptation (e.g., [8]) is that the the level of autonomy is not only dependent on the individual but also on the the other agents being integrated. For instance, in the discussion leader example above, the number of other attendees and their state might be factors in the autonomy decision. Thus, in our approach we emphasize the use of knowledge about other team members (in addition to the preferences of the integrated agent) in using supervised learning techniques. Each proxy learns what decisions it can take autonomously, and what decision need be confirmed with the agent—*in the context of particular scenarios involving other agents*.

Specifically, the Teamcore proxies for humans can make coordination and repair decisions autonomously to aid in team activities like meetings (e.g., the human-collaboration domain). Eleven attributes are used in learning, some of which have been inspired by existing meeting scheduling systems, such as "MeetingMaker", which include meeting location, time, resources reserved etc. However, other attributes describe the state of the other agents participating in the meeting—e.g., the number of persons attending and the most important member attending (in terms of the organizational hierarchy). These attributes are extracted from the user's schedule files, organizational charts, etc. In the training phase, a proxy suggests a coordination decision and a query as to whether the user would wish it to make such a decision autonomously. C4.5[7] is used to learn a decision tree from the interactions with the user.

3.2 Adaptive execution

A proxy's decision, whether autonomous or after consultation with its domain agent, is focused on executing a team activity. Here, the proxies may dynamically adapt their team plans at execution time, based on the performance of member agents. In particular, performance of complex domain agents is likely to vary during the lifetime of the proxy organization. It is thus important that the Teamcore proxies be able to make runtime decisions about plan execution based on the performance of the domain agents. Indeed, the Teamcores can (as a team) dynamically decide whether or not to execute any plans the team programmer marks as optional. Karma gives each Teamcore an initial specification of its domain agent's capabilities, including parameters such as response time (e.g., average, min/max response times are recorded from past runs). However, if the actual runtime performance of a domain agent greatly differs from expectations (e.g., so that the cost in agent response time greatly exceeds the benefits from its results), the Teamcore proxies together modify the optional plans and avoid using this particular domain agent.

Teamcores' plan representation for this decision-making is similar to that used by existing approaches to decision-theoretic planning [1]. The Teamcores begin executing an initial plan sequence that they determine to be optimal given costs and benefits of including the optional plans in the sequence. However, they can dynamically choose to omit optional plans if a particular domain agent's response time should deviate from the expected time cost. For instance, if they had initially decided to include the route planning plan, but the route planner is taking longer than expected, the Teamcores can compare their current plan sequence against alternate candidates, taking into account the increased cost of the current response time. If the time cost outweighs the value of route planning, the Teamcores can change sequences and skip the route-planning step, knowing that they are saving in the overall value of their execution.

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

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

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

3.3 Adaptive monitoring

In addition to executing team activities, Teamcore proxies must also monitor these activities, to detect task execution failures and to allow humans to track the team's progress. To this end, a Teamcore proxy relies on plan recognition to infer the state of its team members from the coordination messages normally transmitted during execution. Such messages do not convey full information about agent state, but can provide hints as to the senders' state. This plan-recognition-based method is non-intrusive, avoiding the overhead of the proxies having to continually communicate their state to other proxies.

Teamcore proxies therefore monitor the communications among themselves. Applying their knowledge of their own communication protocols, the proxies identify exchanges of messages such as those establishing or terminating a team plan. When a plan is terminated/selected, the monitoring Teamcore proxies can infer that execution has reached at least the stage corresponding to the plan. However, in general, every plan may not lead to communication, and hence the plan-recognition process faces ambiguity. For instance, in the evacuation scenario, the Teamcore proxies communicate initially to jointly select the **Obtain-orders** plan. To a monitoring proxy, until a second message is observed, any of the following steps is a possibility. Unfortunately, such ambiguity interferes with monitoring.

To reduce ambiguity in recognized plans, the monitoring system utilizes two adaptation techniques. The first simple technique is to use learning to predict when messages will be exchanged. Such predictions can significantly reduce the number of hypothesized states, since the system knows that the monitored team will not get into certain states without a message being received. At first, an inexperienced system cannot make such predictions. However, as it observes messages being sent, it can construct a model of when such communications will be sent, and use this model to disambiguate the recognized plans. Here, Teamcore currently uses rote-learning successfully; but, other techniques will be investigated in the future.

The second adaptation technique is to have the Teamcore team actively adapt its own behavior to make monitoring less ambiguous. Based on the feedback of the monitoring system, the team of proxies changes its model of communication costs and benefits, so as to ensure that that the proxies communicate at specific points during execution at which ambiguity interferes with the monitoring tasks. For example, when monitoring the evacuation-rehearsal scenario, the human operators often complained that they are unable to distinguish two important states–the state in which the team was flying towards (or from) the landing zone, and the state where the team is carrying out its landing zone operations. When the monitoring system provided this feedback, the proxy team communicated when jointly-selecting the landing-zone maneuvers plan, and the ambiguity in recognized plans was greatly reduced.

3.4 Adjustable information requirements

When executing or monitoring team activities, proxies must also inform the domain agents they represent. However, agents can differ in the amount of information they need in order to successfully carry out their team responsibilities. In the evacuation scenario, the Teamcore proxies sent messages to their domain agents as mandated by the team plans' requirements for tasks and monitoring conditions, without considering communication costs incurred by these messages. More complex agents (including humans) would rather sacrifice some of these messages rather than incur high communication costs. For instance, in our human collaboration scenario, if the system delays a meeting, it can notify the attendees of this delay by sending messages to their PDAs. However, wireless message services usually charge a fee, so some users may prefer not knowing about small delays. The Teamcore proxies should weigh the value of the message (to the user, *as well as to the overall team plan*) against the cost of sending the message to the user.

In addition, heterogeneous agents may have multiple channels of communication, each with different characteristics. In the evacuation scenario, the agents communicated through a single KQML interface. However, with the human agents in our collaboration scenario, the Teamcore proxy can pop up a dialog on the user's screen, send an email message to a PDA (if the user has one), or send email to a third party who could tell the user in person. The dialog box has very little cost, but it is an unreliable means of informing the user, who may not be at the terminal when the message arrives. On the other hand, having a third party tell the user in person may be completely reliable, but there is a high cost.

We can model a communication channel's reliability with a probability distribution over the amount of time it takes for the message to reach the user through that channel. For simplicity, our initial implementation represents this time with an exponential random variable, so that the probability of the message's arriving within time t is $1 - e^{-\lambda t}$, for some reliability parameter λ. We model the cost of the channel and the values of the various messages as fixed values.

Whenever the Teamcore proxy decides to send a message to the user, it first pops up a dialog box on the screen with the message. If the user does not explicitly acknowledge the dialog, the proxy considers using alternate channels. It evaluates the expected benefit of using such an alternate channel by computing the increase in the likelihood of the message reaching the user, based on the comparative reliabilities of any channels used so far and those under consideration. If the product of this increased likelihood and the value of this message exceeds the channel's cost, the proxy sends the message through the new channel.

We have implemented a simple reinforcement learning algorithm to evaluate the communication channel parameters for individual users. For each channel used for a given message, the dialog box on the screen (which is always used during the proxy's learning phase) allows the user to provide feedback about whether the proxy's use of the channel was appropriate and whether the channel transmitted the message to the user in time. Feedback on the former (latter) increments or decrements the channel's cost (reliability) parameter as appropriate.

4 Evaluation

For evaluation, we begin with the evaluation of individual adaptation capabilities, followed by the evaluation of Teamcore's basis on a principled teamwork model, and then of the integrated system. We begin with an evaluation of the adjustable autonomy component of the Teamcore proxies (Section 3.1). Here, we used actual meeting data recorded in users' meeting-scheduling programs, totaling 58 meetings. Five different data sets were constructed out of these, each by randomly picking 36 meetings for training data, and 22 for test data (random sub-sampling holdout). Figure 3 shows the accuracy of the adjustable autonomy prediction plotted against the number of examples used to train the agent (out of the 36 training examples), for the five different data sets. For each data set, we observe that the autonomy learning accuracy increases, usually up to 91%. However, even using more than 36 examples did not improve the accuracy further. In particular, for meetings rated to be of "middle importance", a human's autonomy response some occasions appeared inconsistent, possibly due to actual inconsistency, possibly due to our limited attribute set.

We can also evaluate the benefit of the Teamcores' run-time plan modification capabilities (see Section 3.2). Figure 4 shows the results of varying Ariadne's response times

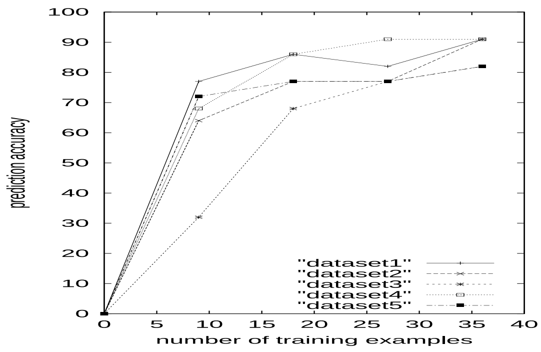

Figure 3. Adaptation of Teamcore autonomy.

on the time of the overall mission execution. In the evacuation plan, Ariadne provides information about missile locations along a particular route. If there are missiles present, the Teamcores instruct the helicopters to fly at a higher altitude to be out of range. The team could save itself the time involved with querying Ariadne by simply having the helicopters *always* fly at the higher safe altitude. However, the helicopters fly slower at the higher altitude, so the query is sometimes worthwhile, depending on the Ariadne's response time. In Figure 4, we can see that when Ariadne's response time exceeds 15s, the cost of the query outweighs the value of the information. In such cases, the Teamcores with the decision-theoretic flexibility skip the query to save in overall execution cost (here, equivalent to time, according to the designer-specified utility function).

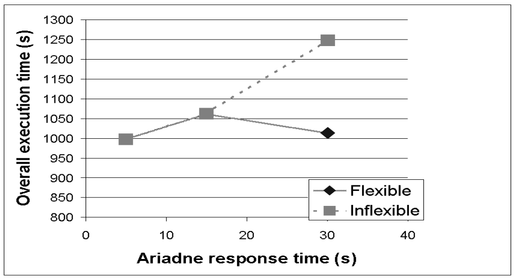

Figure 4. Adapting to variable agent performance.

We have also conducted experiments in adaptive monitoring. Figures 5 presents the results from experiments run in the evacuation scenario with and without the two monitoring adaptation techniques. The X axis notes the observed joint-selections/terminations as the task is executed. Each such observation corresponds to an exchange of messages among the proxies in which they jointly select or terminate a team plan. As execution progresses, we move from left to right along the X axis. The Y axis notes the number of recognized plans based on the current observations, i.e., a higher value means greater ambiguity (worse).

Figure 5 shows the results of learning a predictive model of the communications. We see that without learning, a relatively high level of ambiguity exist which is slowly reduced as more observations are made, and past states are ruled out. However, the system cannot make any predictions about future states of the agents, other than that they are possible. When the learning technique is applied on-line, some learned experience is immediately useful, and ambiguity is reduced somewhat. However, some exchanges are encountered late during task execution, thus they cannot be used to reduce the ambiguity while learning. The third line corresponds to the results when the model has been fully learned. As can be seen, it shows significantly reduced ambiguity in the recognized plans.

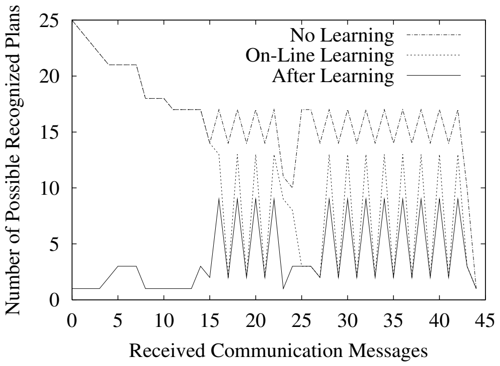

Figure 5. Adaptive monitoring: predictive communication model.

Figure 6 shows the proxy team's adaptation of its communication behaviors significantly reduces the ambiguity in recognized plans, to provide better monitoring. The line marked "Prior to Behavior Adaptation" shows the results of using a fully-learned model of communications to disambiguate recognized plans. The user has decided to disambiguate between the **fly-flight-plan** and the **landing-zone-maneuvers** plans, by causing the agents to explicitly communicate about the joint-selection of the **landing-zone-maneuvers** plan. This corresponds to an additional exchange of messages among the agents (observation 24). This additional observation has an effect much earlier along task execution, greatly reducing the ambiguity beginning with observation 16.

We have also conducted preliminary experiments to evaluate the suitability of our models of the reliability and cost of different communication channels. The results on cost are shown in Figure 7 (the results on reliability are similar and not shown). Here, the users received a series of hypothetical messages and then provided the feedback required for the

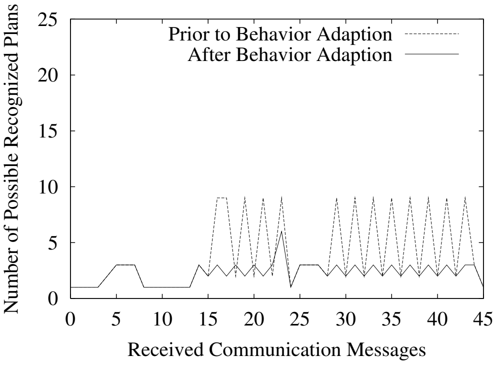

Figure 6. Adaptive monitoring: adpating behavior.

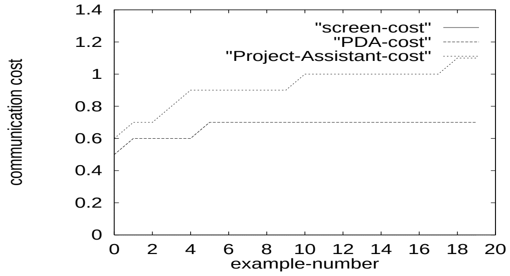

Figure 7. Learning costs of communication channels.

reinforcement learning. Most of the parameters converged monotonically to an equilibrium value, while the cost of the user's screen remained very low. Although it is difficult to gauge the accuracy of the final *values* of the model, the system appeared to make the correct *decisions* about which communication channels to use and when to use them.

We may also attempt to evaluate the benefits of Teamcore proxy's in-built teamwork capabilities. One key alternative to such an in-built teamwork model is reproducing all of Teamcore's capabilities via domain-specific coordination plans. In such a domain-specific implementation, about 10 separate domain-specific coordination plans would be required for each of the 40 team plans in Teamcore[11]. That is, 100s of domain-specific coordination plans could potentially be required to reproduce Teamcore's capabilities to coordinate among each other, just for this domain. In contrast, with Teamcore, no coordination plans were written for inter-Teamcore communication. Instead, such communications occurred automatically from the specifications of team plans. Thus, it would appear that Teamcores have significantly alleviated the coding effort for coordination plans.

While evaluation of the resulting integrated teams is more difficult, it is useful to observe that Teamcore-based system for evacuation rehearsal has often been demonstrated outside our laboratory. Also, the system for human collaboration is being constructed for actual use at our institute.

5 Related Work

In terms of related work, Jennings's seminal work on GRATE*[4] integration architecture is similar to Teamcore, in that distributed proxies, each containing a cooperation module integrate heterogeneous agents. One major difference is that GRATE* proxies do not adapt to individual agents, a critical capability if architectures are to integrate an increasingly heterogeneous, complex agent set. Also, GRATE* cooperation module is arguably weaker than Teamcore's Steam, e.g., Steam enables role substitution in repairing team activity, which is not available in GRATE*.

The Open Agent Architecture (OAA) [6] is an important well known agent integration architecture. OAA (and this family of architectures) provide centralized facilitators to enable agents to locate each other, and a blackboard architecture to communicate with each other, but not teamwork capabilities, or adaptation, as in Teamcore. Also, Teamcore's distributed approach avoids a centralized processing bottleneck, and a central point of failure.

Two other related systems are the RETSINA[10] and IMPACT[9] multi-agent frameworks. While the goals of these frameworks are somewhat similar to ours, their development appears complementary to Teamcore. For instance, RETSINA is based on three types of agents: (i) interface agents; (ii) task agents; and (iii) information agents. Middle agents allow these various agents to locate each other. Thus, as Section 2 discusses, Karma can use RETSINA middle agents for locating relevant agents, while adaptive, infrastructural teamwork in our Teamcores may enable the different RETSINA agents to work flexibly in teams.

Tidhar [14] used the term *team-oriented programming* to describe an implemented framework specifying team behaviors based on team plans, coupled with organizational structures. While Tidhar's work has influenced ours, Tidhar does not address the issue of integrating heterogeneous agents, or architectures for integrating such agents or their adaptation.

6 Conclusion

With software agents, smart hardware devices and diverse information appliances coming into wide-spread use, integration architectures that allow such diverse systems to work together are becoming increasingly important. The need to identify key design principles underlying such architectures is therefore critical. This paper investigates some of these principles through an integrated, adaptive architecture, Teamcore, which is evaluated in two different domains, using a diverse set of software and hardware devices. A key lesson learned from our work is that despite the heterogeneity of agents integrated, sound principles of multi-agent interactions—in our case a principled teamwork model—can serve as a foundation of integration architectures. Such principled foundations can enable rapid development of robust integrated systems. Another key novel lesson is adaptive capabilities are critical in the integration architecture to adapt to the requirements of heterogeneous agents. Adaptation is necessary in different ways, which we demonstrate in four areas: (i) adaptive autonomy; (ii) adaptive execution; (iii) adaptive monitoring; and (iv) adaptive information delivery. There are several avenues for future work, including enhancing the learning mechanisms for quicker and more accurate adaptation.

Acknowledgements

This research was supported by DARPA Award no. F30602-98-2-0108. The effort is being managed by AFRL/Rome research site. We thank Phil Cohen, Katia Sycara and Steve Minton for contributing agents used in the work described here.

References

[1] J. Blythe. Planning with external events. In *Proc. of the Internat'l Joint Conf. on Artif. Intell.*, pages 94–101, 1994.

[2] J. Hendler and R. Metzeger. Putting it all together – the control of agent-based systems program. *IEEE Intelligent Systems and their applications*, 14, March 1999.

[3] M. N. Huhns and M. P. Singh. All agents are not created equal. *IEEE Internet Computing*, 2:94–96, 1998.

[4] N. Jennings. Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. *Artificial Intelligence*, 75, 1995.

[5] N. Jennings. Agent-based computing: Promise and perils. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-99)*, August 1999.

[6] D. L. Martin, A. J. Cheyer, and D. B. Moran. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1-2):92–128, 1999.

[7] J. R. Quinlan. *C4.5: Programs for machine learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[8] S. Rogers, C. Fiechter, and P. Langley. An adaptive interactive agent for route advice. In *Third International Conference on Autonomous Agents*, Seattle,WA, 1999.

[9] T. Rogers, R. Ross, and V. Subrahmanian. Impact: A system for building agent applications. *Journal of Intelligent Information Systems*, October 1999.

[10] K. Sycara, K. Decker, A. Pannu, M. Williamson, and D. Zeng. Distributed intelligent agents. *IEEE Expert*, 11:36–46, 1996.

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

[12] M. Tambe, J. Adibi, Y. Alonaizon, A. Erdem, G. Kaminka, S. Marsella, and I. Muslea. Building agent teams using an explicit teamwork model and learning. *Artificial Intelligence*, 110:215–240, 1999.

[13] M. Tambe, D. Pynadath, and N. Chauvat. Building dynamic agent organizations in cyberspace. *IEEE Internet Computing*, 4(2):65–73, 2000.

[14] G. Tidhar. Team-oriented programming: Social structures. Technical Report 47, Australian Artificial Intelligence Institute, 1993.