Building Dynamic Agent Organizations in Cyberspace

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

January 24, 2000

Abstract

With the promise of agent-based systems, a variety of research/industrial groups are developing autonomous, heterogeneous agents, that are distributed over a variety of platforms and environments in cyberspace. Rapid integration of such distributed, heterogeneous agents would enable software to be rapidly developed to address large-scale problems of interest. Unfortunately, rapid and robust integration remains a difficult challenge.

To address this challenge, we are developing a novel teamwork-based agent integration framework. In this framework, software developers specify an agent organization called a *team-oriented program*. To recruit agents for this organization, an agent resources manager (an analogue of a "human resources manager") searches the cyberspace for agents of interest to this organization, and monitors their performance over time. Agents in this organization are wrapped with TEAMCORE wrappers, that make them *team ready*, and thus ensure robust, flexible teamwork among the members of the newly formed organization. This implemented framework promises to reduce the software development effort in agent integration while providing robustness due to its teamwork-based foundations. A concrete, running example, based on heterogeneous, distributed agents is presented.[1]

[1]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. We also gratefully acknowledge the support of Hank Seebeck of GlobalInfotek. We also thank Mike Huhns for his helpful comments on this work.

1 Introduction

An increasing number of agent-based systems are being developed for complex dynamic environments, such as disaster rescue missions, monitoring and surveillance tasks, enterprise integration, and education and training environments. These agents, in the form of information agents, planning/execution agents, middle agents, user agents or embedded agents must often operate in cyberspace (on the internet or intra−nets) to interface with relevant information sources, network facilities and other agents[3, 4, 1, 12].

This growth in agent-based systems is predicted to be followed by another powerful trend: the reuse of specialized agents as standardized building blocks for large-scale systems[4, 6, 2]. This prediction is based on two observations. First, software systems are being constructed with ever-larger reusable components (rather than as monoliths)[10]. The reuse of agents as components appears to be the next logical step — enabling a richer reuse mechanism than component-ware or application frameworks[6]. Second, we already see the trend of integration of agent components in cooperative information systems, networked embedded systems, and other systems operating in cyberspace[3, 4, 9].

Unfortunately, integrating agents in a large-scale system remains difficult. First, in the distributed, open cyberspace environment, enabling a software developer to locate/recruit relevant agents for integration is a key challenge. Second, since the recruited agents are not likely built to work together, building an integrated system with appropriate inter-agent coordination is difficult. The problem is not only the potentially large amounts of modifications to agent code (to add coordination plans), but also the difficulty of making such modifications in an open, distributed environment, where the agents may be developed by different developers. This problem is further exacerbated if such modifications must be repeated for each new integration. Third, given a dynamic, open environment, the resulting integrated system must be made robust despite uncertainties, such as dynamic agent failures or unavailability (or dynamic availability). While there are other issues in integration as well, such as a common agent communication language, this short article will focus on the above three key challenges.

To address the above agent integration challenges, our TEAMCORE[2] project focuses on enabling software developers to build large-scale agent organizations in cyberspace. There are currently two key aspects to this project. The first focuses on specification and monitoring of the agent organization. The second focuses on enabling the organization to reliably execute tasks, by ensuring robust teamwork among the agents in the organization. The first aspect is the focus of KARMA, the *Knowledgeable Agent Resources Manager Assistant*, which aids a software developer in three ways. First, Karma aids in *team-oriented programming*, i.e., specifying a hierarchical agent organization with roles, and high-level goals and plans for this organization. A key benefit of team-oriented programming is that it abstracts away from coordination details and thus alleviates the difficulty of writing large numbers of coordination plans. Second, Karma locates agents in cyberspace that match the specified organization's requirements and assists in allocating organizational roles to such agents. Third, Karma monitors the organization to enable the developer to diagnose failures and records agents' performance for future (re)organizations. Karma's agent resources management functionality differs significantly from *middle agents* such as matchmakers. Indeed, if the metaphor for middle agents is "middle-men" in physical commerce[1], the metaphor for agent resource managers is "human resources managers" in a commercial firm, and as such, agents such as Karma are expected to be increasingly critical in large-scale agent systems.

Having specified a team-oriented program, the second aspect of the TEAMCORE project focuses on its robust execution. One key hypothesis here is that agent teamwork will enhance robust execution, since as team members, agents can be expected to act responsibly towards each other, cover for each other's execution failures and exchange key information. To enable such teamwork, we make the agents that are assigned roles in the team-oriented program *team-ready*, i.e., responsible team members. An agent's conversion from

[2]TEAMCORE derives its name from its encapsulation of "core team reasoning" as discussed later.

non-team-readiness to team-readiness is accomplished by wrapping the agent with a TEAMCORE wrapper (wrapping avoids the need to modify the agents themselves). The TEAMCORE wrappers are based on the STEAM general-purpose teamwork model[11]. STEAM is a reusable teamwork module that encapsulates reasoning about common teamwork coordination, including contingencies in such coordination. Given this model, the TEAMCORE wrappers automatically generate the required coordination actions in executing a team-oriented program, including recovery from unanticipated failures in its execution, and thus shield the human developer from such specifications. The wrapped team-ready agents can thus robustly execute the team-oriented program specified by the developer.

This paper will describe the Karma-TEAMCORE framework, using a concrete example — evacuation of civilians stranded in a hostile area — where this framework has been successfully applied. In this example, the goal is to build an integrated system for mission rehearsal (in simulation). The 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. Our framework enabled such an integrated system to be built from 11 different existing agents, including a multi-modal user interface agent, a route-planner, a web-querying information-gathering agent and synthetic helicopter pilots. These agents, developed by different developers, were written in four different computer languages, ran on two different operating systems, and were distributed geographically. Here, Karma was used to specify the necessary team-oriented program, and to successfully locate relevant agents by interfacing with a matchmaker and other middle agents. The team-oriented program is successfully executed by agents wrapped with TEAMCORE wrappers. This execution is shown to be robust against failures of either the individual agents or the wrappers.

2 Karma-TEAMCORE Framework

Figure 1 shows the overall Karma-TEAMCORE framework to build agent organizations. The numbered arrows show the typical stages of interactions in this system. In the first stage, human software developers interact with TOPI (team-oriented programming interface) to specify a team-oriented program, with an organization and its goals and plans. TOPI in turn interacts with Karma (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 these domain agents are located, Karma aids a developer in assigning agents to organizational roles. Having thus fully defined a team-oriented program, Karma launches the TEAMCOREs. Each TEAMCORE wraps an individual agent assigned in the organization, and the TEAMCOREs jointly execute the team-oriented program, as a team. While executing this program, the TEAMCOREs broadcast information among themselves via multiple broadcast nets — a TEAMCORE sends a message for a team to a broadcast net, which forwards it to all its teammates (stage 4). TEAMCOREs also communicate with the domain agents (stage 5). Karma also "eavesdrops" on the various broadcasts to monitor the progress of TEAMCOREs (stage 6), which it displays to the software developer for debugging. All communication among TEAMCOREs, between a TEAMCORE and its wrapped agent and with Karma occurs via the KQML agent communication language.

A key novelty and strength of our framework is that powerful teamwork capabilities are built into its foundations, i.e., in the TEAMCORE wrappers. These wrappers enable agents that were not originally built to team up, to plan/act in a team. Thus, the team formed with TEAMCORE wrappers can execute a team-oriented program, where agents automatically cover for failed teammates, supply key information to help each other, etc. This framework strongly contrasts with previous agent integration frameworks such

as the Open Agent Architecture (OAA) [7] that provide centralized facilitators to enable agents to locate each other, but not teamwork capabilities. Also, TEAMCORE's distributed approach avoids a centralized processing bottleneck, and a central point of failure.

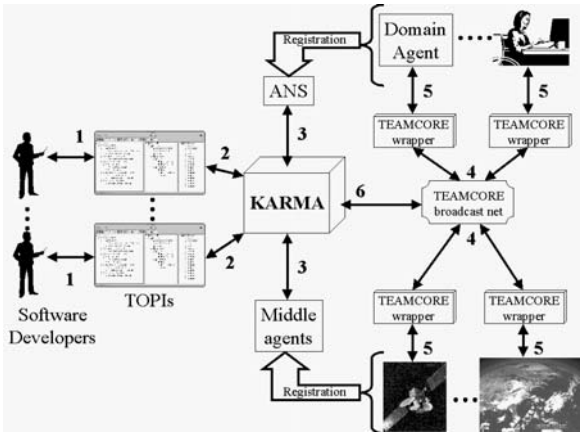

Figure 1: The overall Karma-TEAMCORE framework. TEAMCOREs may wrap different "domain agents" which may include information gathering agents, user assistants etc.

3 Specifying and Monitoring Team-Oriented Programs: Karma

To describe our JAVA-based Karma agent, this section describes the three key ways in which Karma helps a software developer in building an agent organization: (i) Specifying a team-oriented program; (ii) Locating relevant agents in cyberspace and assisting in allocation of roles to such agents; (iii) Monitoring and recording agents' performance for diagnosis.

3.1 The Team-Oriented Program

A software developer begins specifying an organization of interest via "team-oriented programming", which implies that the developer specifies three key aspects of a team: (i) a team organization hierarchy; (ii) a team (reactive) plan hierarchy; and (iii) assignments of agents to execute plans. The team organization hierarchy consists of roles for individuals and for groups of agents. For example, Figure 2-a illustrates a portion of the organization hierarchy of the roles involved with the evacuation scenario from Section 1. 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.

The second aspect of a team-oriented program involves 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 they explicitly express joint activities of the relevant team. 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, conditions which cause the reactive team plan to be achieved, irrelevant or unachievable; and (iii) team-level actions to be executed 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

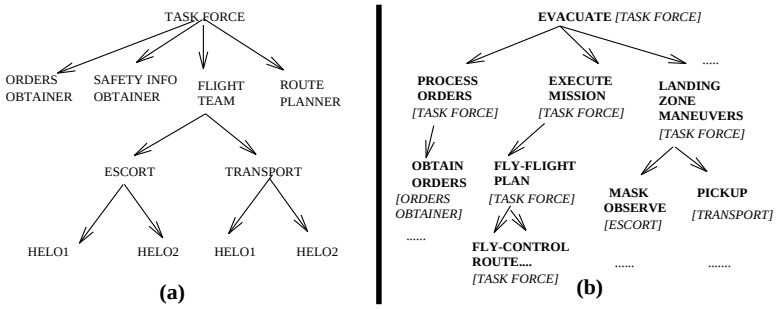

Figure 2: (a) Partial organization hierarchy with roles; (b) Partial reactive team plan hierarchy, both for the evacuation scenario.

Obtain-orders is left unspecified — thus both simplifying the specification while allowing for the reuse of different agents to execute this plan (as shown below).

The software developer must also specify any domain-specific coordination constraints in the execution of team plans. In the example program in 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 represent the domain-specific constraint that a subteam assigned to perform **Pickup** cannot do so until the other subteam assigned **Mask-Observe** has reached its masking locations and begun observing.

The third aspect of team-oriented programming is assignments 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 ommitted from 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, no synchronization actions are programmed — instead, during execution, sychronization actions among members of *Task Force* are automatically enforced, both with respect to the time of plan execution and the identity of the plan (i.e., all members will choose the same plan out of a set of multiple candidates). Similarly, there is no need to specify the coordination actions for coherently terminating reactive team plans; such actions are automatically executed. Domain-specific coordination constraints, such as the one between **Mask-Observe** and **Pickup** discussed above, are also automatically enforced. This automatic generation of coordination actions is due to the TEAMCORE wrappers, as discussed later.

To facilitate encoding of the team-oriented program, Karma interacts with a developer via the TOPI interface. Figure 3 shows a sample screenshot 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 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 such as **Mask Observe** are its coordination constraints, preconditions, assigned subteam, and so on. TOPI also facilitates the assignment of domain agents to roles, and this will be discussed in the next section.

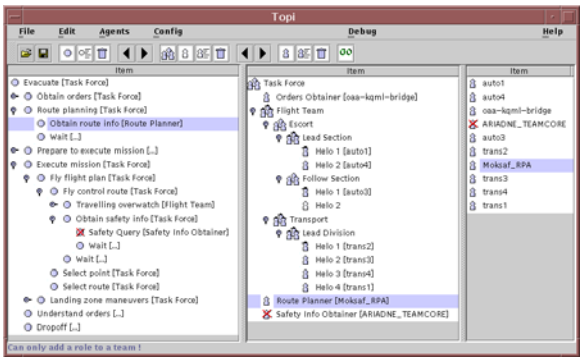

Figure 3: TOPI: The team-Oriented programming interface.

3.2 Searching and Assigning Agents

As discussed previously, Karma derives requirements for individual roles in the organization based on their assignment to plans. Based on these requirements of the organization, Karma searches for agents in cyberspace with matching capabilities. Limiting the search for available agents to just the organizational requirements avoids overwhelming the software developer with too much unnecessary information.

To this end, Karma searches multiple sources: middle agents like matchmakers, known agents from local white pages or agent naming service (ANS) and other registry services. With respect to matchmakers, Karma exploits their powerful matching functionality — rather than replicating all such functionality — to search for relevant agents. For instance, Karma may query AMatchMaker [1] middle agent from Carnegie Mellon University over the internet. AMatchMaker accepts KQML messages specifying an advertisement template and returns a list of descriptions of registered agents that match the template (or that are close enough to be considered relevant). In addition, Karma maintains a database of agents and capabilities, which it searches for names of agents that match the requirements of the organization. It then checks white page services for such agents — here, Karma mainly checks that these agents are currently registered. More recently, Karma has also been interfaced with "the Grid" (produced by DARPA's COABS program[2]), which provides registration and other interconnection services based on JAVA *remote method invocation* (RMI) and SUN's JINI mechanisms. Karma uses RMI to query the Grid registry, to get a list of agents whose entries in the registry match the organizational requirements. (In some cases, since a single agent may be registered with multiple middle agents, its duplicate registration has to be filtered by comparing agents' name and address.)

Thus, from these different sources, Karma compiles a list of relevant agents, and their properties, including address (host and port), capabilities and other miscellaneous information from the source. Karma's database may already have some information, such as reliability, about the agent from previous runs. From this list of relevant agents (in the right pane of TOPI in Figure 3), the developer can assign agents to the roles in the specified organization. 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.

3.3 Monitoring and Recording Agent Performance

While the team executes its program, Karma's task shifts to monitoring and recording this execution, for feedback to the developer. Currently, Karma's observations are non-intrusive, based on eavesdropping on the multicast messages that the TEAMCORE wrappers send as part of their normal operation. Karma analyzes critical messages (e.g., a TEAMCORE's expressing a commitment to a team plan), and each such

messages triggers feedback in TOPI, allowing the developer to see which team plans the team members are executing.

To facilitate the reuse of agents for multiple tasks (or multiple runs), Karma also records how well agents performed in the current task, to aid the developer in the next design task. Thus, a TEAMCORE wrapper sends information regarding the wrapped agent to Karma, e.g., any catastrophic failures during a run, success during runs, response times etc. This information is maintained in Karma's local database. In addition, information such as catastrophic failure is immediately displayed on TOPI to aid debugging. For instance, in Figure 3, TOPI shows the developer that Ariadne is disabled, and that a key role is thus not executed. Karma currently also logs all of the messages it monitored, for possible examination for interesting events.

4 Executing Team-Oriented Programs

While Karma is used to specify the team-oriented program and monitor its execution, the program is executed by a distributed set of TEAMCORE wrappers based on the Soar [8] rule-based integrated agent architecture. Each wrapper contains the STEAM teamwork model, responsible for TEAMCORE's teamwork reasoning and an interface module, each written as a set of Soar rules. We first briefly describe STEAM and then the interface.

4.1 STEAM Component of TEAMCORE

The STEAM teamwork model[11] encodes domain-independent rules that explicitly outline team members' responsibilities and commitments in teamwork. In essence, it attempts to provide agents with "common-sense" knowledge of teamwork to enable them to autonomously reason about coordination and communication. We can divide STEAM's teamwork knowledge into three classes of domain-independent teamwork rules. The first class consists of *coherence preserving* (CP) rules that require team members to communicate with others to ensure coherent initiation and termination of team plans. For instance, if an agent privately discovers that the currently selected team plan is unachievable, then it must not unilaterally terminate the plan. Instead, it must communicate this unachievability information to its teammates, so that they do not waste their resources and that the team as a whole coherently terminates the team plan. As an example of STEAM's application, consider a simulated helicopter team flying in formation. If one of the helicopter pilots observes unanticipated enemy vehicles on their flight route, STEAM's CP rules require that this pilot inform its teammates about the enemy units (since it makes the relevant team plan unachievable), so that the team reacts coherently to the enemy.

Monitor and repair (MR) rules detect if a team task is unachievable due to unexpected member (individual or subteam) failure. It then leads the team into reorganization—via reassignment of roles—to overcome this failure. For example, if a software failure causes an agent to become unavailable, then the MR rules would allow the team members to first check if their current plan is unachievable. If so, the rules also trigger the reorganization of the team, to allow it to continue pursuing the plan. STEAM requires *selectivity-in-communication* (SC) rules because attaining perfect coherence in teamwork can sometimes require excessive communication. STEAM avoids the potentially detrimental side effects of such communication through decision-theoretic communication selectivity. For instance, in some cases, if there is high likelihood that an agent's teammates can obtain the relevant information via their own observation (even if with a slight delay), then it can avoid costly communication.

4.2 TEAMCORE Extensions to STEAM

In the original STEAM implementation, the teamwork knowledge resided directly in the domain agent's knowledge base, which is difficult in an open, heterogeneous environment. By placing this knowledge (rules) in a separate TEAMCORE wrapper, we now no longer need to modify code in the domain agent. The wrappers use KQML for all inter-agent communication. Thus, the wrapped domain agents must provide a KQML interface to communicate with TEAMCOREs.

In addition to the STEAM rules, a TEAMCORE wrapper also contains an interface module to interface with the domain agent. In particular, the STEAM rules enable the TEAMCORE wrappers to communicate with each other automatically to maintain team coherence and recover from member failures. In contrast, the interface module enables a TEAMCORE wrapper to communicate with the domain agent it wraps, to send it tasks and monitoring requests.

With respect to monitoring, a key issue here is that it is the domain agent and not the TEAMCORE wrapper, that often has access to information relevant to the achievement, irrelevance, and unachievability of the team plans (e.g., an information-gathering agent can search in cyberspace for known threats to a team of helicopters). Yet, knowledge of the current team plans is with the TEAMCORE wrapper, so the domain agent may not know what observations are relevant (e.g., threats to the helicopter team are relevant), necessitating communication regarding monitoring. For each team plan, the TEAMCORE wrappers already maintain the termination conditions — conditions that make the team plan achieved, irrelevant, or unachievable. Each TEAMCORE wrapper also maintains a specification of what its domain agent can monitor. Thus, if a domain agent can monitor conditions that reflect the achievement, irrelevance, or unachievability of a team plan, then the TEAMCORE wrapper automatically requests it to monitor any change in those conditions. The response from the domain agent may be communicated with other TEAMCORE wrappers, as part of the usual STEAM procedures.

4.3 Example: Evacuation Scenario

We have applied our Karma-TEAMCORE framework to the problem of mission rehearsal of the evacuation of civilians from a threatened location (discussed in Section 1). The software developer was able to create a team-oriented program for this problem, and the following agents were used:

Quickset: (developed by P. Cohen et al., Oregon Graduate Institute) A human operator uses speech and gesture to provide the command to evacuate, as well as the locations of the evacuation site and other meaningful route landmarks. Quickset uses a multimodal input device to interpret the operator's instructions. [Written in C++, runs on Windows NT operating system]

Route planner: (developed by Sycara et al., Carnegie-Mellon University) Given source and destination locations and a map of the region, produces a detailed route for aircraft, avoiding known obstacles (e.g., no-fly zones). [Written in C++, runs on Windows NT operating system.]

Ariadne: (developed by Minton et al., USC Information Sciences Institute) Queries a database on dynamic obstacles and threats (e.g., enemy missile launchers). [Written in Lisp, runs on Unix platforms.]

Helicopter pilots: (developed by Tambe, USC Information Sciences Institute) These pilot agents were earlier developed for distributed interactive simulations[12], and they can fly helicopters along a specified route, land at specified destinations, and mask for observations. The pilot agents used in this work have no teamwork capabilities. [Written in Soar, run on a Unix Platform.]

As seen above, these 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 have no pre-existing teamwork capabilities. There are actually 11 agents overall, including the Ariadne, route-planner, Quickset, and eight different helicopters (some escorts and some transports). These agents provided a fixed specification of possible communication and task capabilities.

The Karma-TEAMCORE framework was used to successfully build a team-oriented program for evacuation mission rehearsal system from these agents. Karma can locate these agents based on the team-oriented program and the specified organization hierarchy (although, at this stage, collaboration was pre-arranged with other researchers). The team-oriented program was then successfully executed by the TEAMCORE wrappers. This program had 18 reactive team plans, which all of the TEAMCORE wrappers loaded. Successful execution of this program required the TEAMCOREs to appropriately communicate with other TEAMCOREs, and to generate the correct tasking and monitoring requests for the domain agents.

There are several aspects to the evaluation of the Karma-TEAMCORE framework. One key aspect of evaluation is robustness, given that is one of the motivations for the teamwork foundations of TEAMCORE. Here, we have tested TEAMCORE by forcing individual TEAMCOREs or agents to fail, e.g., by causing the TEAMCORE wrapping a helicopter pilot or an agent such as the route-planner to crash. Of course, such episodes arise "naturally" as well. In general, our observation is that the system does not come to a halt; instead, the teammates (rest of the TEAMCOREs) try to substitute another agent with relevant expertise if necessary (the MR axioms of STEAM) and/or show graceful degradation. For instance, if a TEAMCORE wrapper crashes, another TEAMCORE wrapper, which wraps an agent with similar capabilities takes over. Similarly, if the route-planner crashes, the system reverts back to using straight-line paths. The failure recovery is not perfect however — agents are on rare occassion unable to detect that one of them has failed, and hence are unable to launch the failure recovery operations. Improving distributed monitoring remains an issue for future work.

A second aspect of evaluation is measuring the benefit of the TEAMCORE wrappers' capabilities, in particular, its domain-independent teamwork knowledge, by comparing with alternatives. One key alternative 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 18 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 special "wiring" was required, i.e., 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.

Figure 4 shows the number of messages exchanged over time in different runs. The x-axis measures the time elapsed while the y-axis shows the cumulative number of messages exchanged on a *log scale*. The "normal" run shows the number of messages typically exchanged among TEAMCOREs for the evacuation scenario with time. The key here is that these approx 100 messages are automatically generated by the TEAMCOREs. The "cautious" run shows the number of messages exchanged among TEAMCOREs without the decision theoretic communication selectivity in STEAM, illustrating both the overhead reduction via such reasoning, and the difficulty of hand-coding coordination plans (simple plans may lead to significant overheads). Finally, the "failure" run shows the messages exchanged among TEAMCOREs if the Ariadne agent were to crash unexpectedly (in the course of a "normal" run). To compensate for such failure, there is an initial increase in the total messages; but once the failure is compensated for, fewer messages are exchanged, so that the total messages in the "failure" and "normal" runs is roughly the same.

A third aspect of evaluation is the ease of modifications to the team. Our framework appears to facilitate changes to the team, at least compared with the alternative of domain-specific coordination. For instance, the route planner was the last addition to the team. The pre-existing team would successfully execute straight-line routes based on the human commander's orders. We wanted to add the route planner to specify routes around potential obstacles. We first constructed a TEAMCORE wrapper to make the route planner team-ready. However, this wrapper was the same as all of the existing wrappers, so no additional coding was necessary to add the general-purpose teamwork model of TEAMCORE, nor the domain-specific team

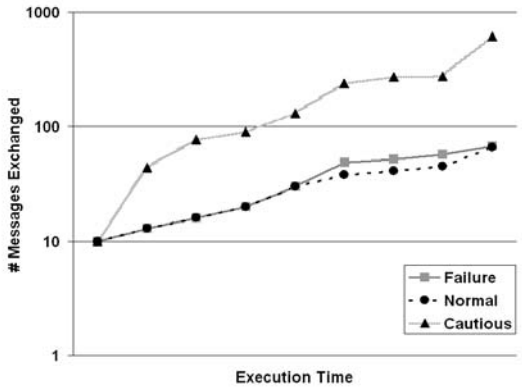

Figure 4: Comparison of messages exchanged in execution.

program. To extend the organizational hierarchy, we simply added the route planner as a member of *Task Force*. We then added the **Process-Routes** branch of the plan hierarchy to allow for route planning. This branch involves very simple plans where the TEAMCORE wrapper 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.

Finally, as discussed in Section 1, this short paper clearly did not intend to address all of the issues in agent integration. Thus, one weakness of this work is that issues such as a common agent communication language, a common ontology and so on, are not addressed in our framework.

5 Related Work

In terms of related work, we have discussed TEAMCORE's relationship with centralized integration architectures such as OAA[7] in Section 2. Another related system is the RETSINA multi-agent infrastructure framework[9], which is based on three different types of interacting agents: (i) interface agents that interact with users; (ii) task agents that perform tasks; and (iii) information agents that interact with information sources. Middle agents are used to enable agents to locate each other. This effort appears quite complementary to TEAMCORE. Indeed, as discussed earlier, RETSINA middle agents could be used by Karma for locating relevant agents, while infrastructural teamwork in TEAMCOREs may aid the different RETSINA agents to work in teams.

Tidhar [13] has used the term *team-oriented programming* to describe an implemented framework specifying team behaviors based on team plans, coupled with organizational structures. A team plan is automatically unfolded into plans for individual agents containing communicative acts which ensure rudimentary coordination. While many features of Tidhar's framework are important in the context of TEAMCORE, the critical issue of agent reuse, particularly involving heterogeneous agents, is not addressed. Thus, agent resources manager such as Karma for searching agents and monitoring their performance, wrapping technology that involves (at least) the addition of tasking and monitoring capabilities in the teamwork layer are novel in our framework. Furthermore, the flexibility of team-reorganization on failure does not seem to be part of the abstract team-layer of Tidhar's framework.

Jennings's early work on GRATE*[5], also based on a teamwork model, is also relevant to our framework. However, TEAMCORE's STEAM model allows teamwork to a deeper level than the single joint goal and

plan allowed in GRATE*, and STEAM also provides capabilities for monitoring role-performance and role substitution in repairing team activity, which is not available in GRATE* Furthermore, the notions of building team-oriented programs to specify agent organizations in cyberspace, agent resources manager to aid in building and monitoring such programs, are not addressed in GRATE*.

6 Conclusion and Future Work

Rapid integration of heterogeneous, distributed sets of agents would enable new capabilities that are currently beyond the reach of software designers. To this end, this article focused on enabling designers to rapidly create agent organizations in cyberspace. It described an agent resources manager, Karma, for assistance in effectively creating and managing agent organizations. Indeed, one key take-away lesson of this work is that as the number and variety of agents in cyberspace increases, new types of agents that aid in building and maintaining agent organizations would become increasingly critical.

Another key take-away lesson from this work is that building powerful, reusable multi-agent interaction techniques into the integration infrastructure itself facilitates rapid integration. Indeed, such an approach can improve robustness and flexibility of the resulting system. In particular, a key novelty of the Karma-TEAMCORE framework is that teamwork capabilities are built into its very foundations, in the form of the teamwork models in our TEAMCORE wrappers. These wrappers make existing individual domain agents, who are originally not ready to be responsible team members, "team ready". Once made team-ready, these agents enable abstract specifications of an agent organization in the form of team-oriented programs. This can significantly reduce the software design effort, since team-oriented programs eliminate the need to script all of the agent interactions.

Our framework has shown promise, given its successful application in a concrete example of the evacuation scenario. Team-oriented programming was used to correctly task a set of heterogeneous, distributed (and originally non-team-ready) agents. The team formed from these agents by TEAMCOREs successfully executed the team-oriented program, and the team is robust enough to survive despite individual failures. Our on-going work is focusing on providing more of the "human resources" type capabilities to Karma, e.g., to hire agents for some probationary period in a team, to fire them or to explain to the software developer that such firing was justified (without any unfair bias). We are also beginning to collaborate with other researchers to address issues in common ontologies and languages.

References

[1] K. Decker, S. Sycara, and M. Williamson. Middle-agents for the internet. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-97)*, July 1997.

[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. Networking embedded agents. *IEEE Internet Computing*, 3:91–93, 1999.

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

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

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

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

[8] A. Newell. *Unified Theories of Cognition*. Harvard Univ. Press, Cambridge, Mass., 1990.

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

[10] C. Szyperski. *Component software: Beyond object-oriented programming*. Addison Wesley, Menlo Park, CA, 1999.

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

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

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