# Toward Team-Oriented Programming

David V. Pynadath, Milind Tambe, Nicolas Chauvat[***], Lawrence Cavedon[†]

Information Sciences Institute and Computer Science Department
University of Southern California
4676 Admiralty Way, Marina del Rey, CA 90292
{pynadath,tambe,nico,cavedon}@isi.edu

**Abstract.** The promise of agent-based systems is leading towards the development of autonomous, heterogeneous agents, designed by a variety of research/industrial groups and distributed over a variety of platforms and environments. Teamwork among these heterogeneous agents is critical in realizing the full potential of these systems and scaling up to the demands of large-scale applications. Unfortunately, development of robust, flexible agent teams is currently extremely difficult. This paper focuses on significantly accelerating the process of building such teams using a simplified, abstract framework called *team-oriented programming* (TOP). In TOP, a programmer specifies an agent organization hierarchy and the team tasks for the organization to perform, abstracting away from the innumerable coordination plans potentially necessary to ensure robust and flexible team operation. Our TEAMCORE system supports TOP through a distributed, domain-independent layer that integrates core teamwork coordination and communication capabilities. We have recently used TOP to integrate a diverse team of heterogeneous distributed agents in performing a complex task. We outline the current state of our TOP implementation and the outstanding issues in developing such a framework.

## 1 Introduction

Agent-based systems currently operate in complex, dynamic environments such as user interfaces [18], robotic space missions, virtual training environments [22], and Internet information extraction [28]. These agents are often autonomous, heterogeneous, and distributed over a variety of platforms and domains. Yet, users may desire such diverse agents to work together to accomplish novel, complex tasks. Such reuse of existing agents is preferable to building a monolithic application from scratch, as it promises to significantly reduce software development effort while preserving modularity.

Indeed, agents working together in teams can tackle user-defined tasks more complex than those they can perform as individuals. However, constructing such teams remains a difficult challenge. In particular, current approaches to designing agent teams lack the general-purpose teamwork models that would enable agents to autonomously reason about the communication and coordination required in teamwork. The absence of such teamwork models makes team construction highly labor-intensive. In particular,

[***] Visiting from PSA Peugeot-Citroën, Centre de Recherche de Vélizy-Villacoublay, France
[†] Visiting from the Computer Science Dept., RMIT, Melbourne, Australia

to enable agents to autonomously reason about coordination, human developers must provide them with large numbers of domain-specific coordination and communication plans. These domain-specific plans are not reusable, so we must develop new ones for each new domain. Furthermore, the resulting teams often suffer from a lack of robustness and flexibility. In real-world domains, teams face a variety of uncertainties, such as a team member's unanticipated failure in fulfilling responsibilities, members' divergent beliefs about their environment, and unexpectedly noisy or faulty communication. Without a teamwork model, it is difficult to anticipate and pre-plan for the vast number of coordination failures possible due to such uncertainties.

This paper focuses on minimizing the complexity of building robust and flexible teams via a domain-independent infrastructure to support *team-oriented programming* (TOP). In our proposed view of TOP, a "team-oriented programmer" has a set of (possibly heterogeneous) agents available. One builds a team to accomplish a task, not by building large numbers of coordination plans, but rather by representing only the domain-specific knowledge about team plans, as well as the organization hierarchy of the existing agents that are intended to execute the team plans. The TOP infrastructure then automatically ensures agents' coordinated commitment to team plans, maintenance of coherent group beliefs, appropriate tasking of individuals, and reorganization when teammates are unable to perform their assigned tasks. We can reuse this infrastructure even as we change the agents, the tasks, or even the overall problem domain. Section 2 describes our proposed view of TOP in more detail.

We have recently completed an initial implementation of a TOP infrastructure. Our software system, TEAMCORE, integrates a general-purpose teamwork model and provides core teamwork capabilities to individual agents by wrapping them with TEAMCORE, as described in Section 3. Here, we call the individual TEAMCORE "wrapper" a TEAMCORE agent. A TEAMCORE agent is a purely social agent, in that it has only core teamwork capabilities, e.g., it does not possess sensing or action capabilities in the domain of interest. We can take an existing agent that *does* have domain-level action capabilities and make it *team-ready* through an interface with a TEAMCORE agent. Agents made team-ready can rapidly assemble themselves into a team in any given domain. Unlike past approaches, such as the Open Agent Architecture (OAA) [15], which provides a centralized blackboard facilitator to integrate a set of agents, TEAMCORE is a fundamentally distributed team-oriented system. Furthermore, unlike OAA, TEAMCORE allows direct tasking at the team level via team-oriented programming.

We have applied our current TOP infrastructure toward a concrete problem, the evacuation of civilians stranded in a hostile area. In this target scenario, we wish to build a system to enable a set of helicopters to fly in a coordinated formation to a landing zone, pick up stranded civilians, and then fly back to a safe area. The system should enable a human commander to interactively provide the helicopters with locations of the landing zone, the safe area, and other key points in the evacuation route. Furthermore, the system needs to plan routes to avoid known obstacles, to dynamically obtain information about enemy threats, and to change routes when needed.

In this evacuation scenario, TOP has been able to successfully integrate at least eleven different agents into a team. Four of these agents are escort helicopter pilots, and four are transport helicopter pilots, while the rest include a diverse set of agents

created by different developers: a multi-modal user interface agent (Quickset) [7], a route-planning agent, and an information-gathering agent (Ariadne) [13]. Quickset is itself a collection of agents, but our framework treats it as a single agent. These agents are written in four different computer languages, run on two different operating systems, and distributed geographically, yet TOP enables teamwork among them all.

## 2 Team-Oriented Programming

### 2.1 Agent Capability Descriptions

In our framework, the team-oriented programmer must develop a system to perform a joint task. The programmer has a pool of available agents, possibly developed by different designers, based on different agent architectures, implemented in different languages, and running on different operating systems. Following the software engineering trend of reuse, we would like to reuse these agent systems without modification in implementing the team. We assume that each agent that is a potential team member has a functional interface describing its *capabilities*. The capability description for an agent specifies: (i) the set of tasks that the agent can perform; (ii) the input parameters for the task request, as well as constraints on these parameters; (iv) outputs from performing the requested task, as well as constraints on the output. Figure 1 provides a partial capability description for some of the agents in the evacuation scenario. While this capability description outlines key features and the relevant syntax, there is clearly a need for a common ontology or a translation mechanism to provide clear semantics for this description. This issue is outside the scope of this paper.

```
(Route-Planner
    (goal plan-route
        (input (Start-point (unit latitude-longitude))
               (End-point (unit latitude-longitude)))
        (ouput (List-of-points (list-of (unit latitude-longitude)))))
)

(Helicopter
    (goal fly-coordinated
        (input (Distance (unit meters))
               (Angle (unit degrees))
               (Relative-altitude (unit feet)))
    (goal monitor-helicopter
        (input (Vehicle (type helicopter)))
        (output (Status (type helicopter-status))))
   (goal monitor-ground-unit
        (input (GUnit (type ground-unit)))
        (output (Status (type ground-unit-status))))
   (goal monitor-location
        (input (Vehicle (type helicopter)))
        (output (Location (unit latitude-longitude))))
   (goal monitor-distance
        (input (Vehicle (type helicopter)))
        (output (Distance (unit meters))))
)

(Ariadne
    (goal provide-safety-info
        (input (North-west-point (unit latitude-longitude))
               (South-east-point (unit latitude-longitude)))
        (output (List-of-hazards (list-of (unit latitude-longitude)))))
)

(QuickSet
    (goal provide-nav-plan
        (output (Start-point (unit latitude-longitude))
                (End-point (unit latitude-longitude))))
    (goal provide-labeled-points
        (output (List-of-points (list-of (unit latitude-longitude))))
    (goal provide-number-of-units
        (output (number-of-transport (unit number))
                (number-of-escorts (unit number))))
)
```

**Fig. 1.** A partial description of capabilities for some of the agents in the Evacuation scenario.

Figure 1 shows that the route-planning agent can perform the **Plan-Route** task, where the request message must provide the start and destination (in latitude-longitude coordinates) as input. The output is a route, i.e., a list of points. A helicopter pilot agent can perform several tasks, including coordinated flight, i.e., following another aircraft. The parameters here include the distance, angle, and altitude at which to follow the aircraft. The constraints on the input specify the units of those parameters. The helicopter pilot agent can also perform the task of monitoring, by observing features of its own vehicle, the terrain, or other agents. For a pilot agent to monitor the status of another helicopter, the parameters must identify the specific vehicle and the attribute to be monitored (status), and the output is the value of the specified attribute.

Thus, the tasks may have varied types. They may involve achievement goals, as in the route-planning example above, or maintenance goals, as in the case of a helicopter pilot's coordinated flight. Furthermore, tasks may initiate an activity, as in the examples above, or tasks may also terminate activities. For instance, a task may request that an agent terminate a task previously initiated. Tasks may also extend beyond active achievement of goals to include passive observation of conditions that potentially affect team execution, as in the monitoring tasks discussed above.

### 2.2 The Team-Oriented Program

Given this pool of agents, the team-oriented programmer's job is to implement the problem-specific aspects of the system. We can specify these problem-specific aspects at the "team level", a level of abstraction requiring that the programmer specify only:

- the team organization hierarchy for achieving the team goals
- the team goals and the team procedures for achieving them, including:
  - models of initiation conditions, when the team should propose the goal
  - models of conditions for achievement, irrelevance, and unachievability, when the team should terminate the goal
- coordination constraints among the agents executing the team's joint activities

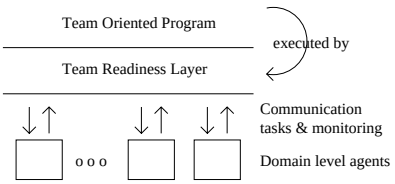


**Fig. 2.** An abstract view of team-oriented programming.

This team-oriented program is to be executed via a TOP infrastructure, which consists of a "team-readiness" layer that provides an interface for agents written in different language. Figure 2 illustrates our view of the TOP infrastructure that enforces team behavior. It ensures coherent execution of team plans by providing task instructions to

the domain-level agents at the appropriate times. This team-readiness layer hides the details of the coordination behavior, low-level tasking, and flexible re-planning. We can implement this abstract specification of the TOP infrastructure in several ways, and we make no assumptions about its internal structure (e.g., whether it consists of a set of distributed agents or a single controller).

The team-oriented program specifies the organization hierarchy through roles that may be filled by individual agents or groups of agents. Figure 3 illustrates a portion of the evacuation scenario's organization hierarchy, where each leaf node corresponds to an individual role, while the internal nodes correspond to teams of agents. At each of these nodes, we have a description of the required capabilities of the corresponding agent or subteam. For instance, the "Orders-Obtainer" role requires an ability to acquire knowledge of the mission parameters. The labels in italics specify the domain-level agent currently filling the corresponding role within the organization.

We currently implement the team goals and procedures aspect of the team-oriented program via reactive team plans. While these reactive team plans are much like situated plans or reactive plans for individual agents [8], the key difference is that they explicitly express joint activities of the relevant team. These reactive team plans ensure that all TEAMCORE agents know the overall team procedure. This team procedure may execute a team activity, plan a team activity, collaboratively design an artifact, collaboratively schedule, or collaboratively monitor and diagnose.

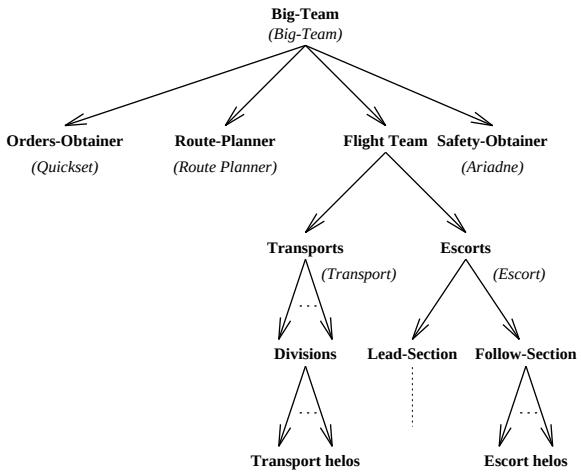


**Fig. 3.** Partial organizational hierarchy of agents in evacuation team.

The reactive team plans in the team program require that the programmer specify the initiation conditions, termination conditions, and team-level actions to be executed as part of the plan. The TOP infrastructure ensures that the team will synchronize itself appropriately in executing the plan. Thus, the programmer need not program such synchronization actions. The infrastructure ensures such synchronization with respect

to both time of plan execution and the identity of the plan, so all members will choose the same plan out of a set of multiple candidates.

The programmer also specifies the termination conditions, under which a team plan is achieved, irrelevant or unachievable. Such explicit specification ensures that *all* team members have this knowledge, so that the team can terminate the goal coherently. Once again, the programmer does not specify the procedure for coherent plan termination. Instead, the TOP architecture uses the termination conditions as the basis for automatically generating the communication necessary to jointly terminate a team plan.

Figure 4 shows some of the team goals and procedures for the evacuation domain. Each nodes corresponds to a goal (in bold), as well as the agent or team (in parentheses) responsible for its achievement. The team programmer specifies particular roles or subteams, taken from the organization hierarchy, to perform a given goal. The TOP infrastructure then assigns agents/subteams to the appropriate roles. For instance, it could assign Quickset the role of "Orders-Obtainer" by noting that Quickset can achieve the **Obtain-Orders** goal. The links of Figure 4 correspond to decomposition and abstraction relationships between goals and subgoals.

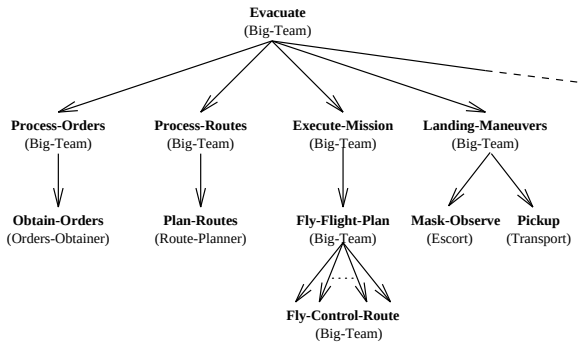


**Fig. 4.** Partial reactive plan hierarchy for evacuation scenario.

With respect to the actions of the reactive team plans, **Evacuate** decomposes into a sequence of subgoals beginning with **Process-Orders**, where the agents of Big-Team must interpret orders provided by a human commander. The team achieves **Process-Orders** by having the agent or subteam labeled "Orders-Obtainer" achieve the **Obtain-Orders** goal. The programmer does not specify the actions of the **Obtain-Orders** plan, instead assuming that the Orders-Obtainer agent/subteam knows how to achieve the **Obtain-Orders** goal. This level of abstraction allows the team-oriented programmer to ignore the inner workings of the member agents, thus simplifying the specification task while also allowing for the reuse of team-oriented programs with different collections of agents. The abstraction also allows the programmer to omit explicit tasking of the domain-level agents. Instead, the TOP architecture automatically generates the appropriate tasking messages at run time.

The programmer must also specify any coordination constraints in the execution of team goals. In the example program, the goal **Landing-Maneuvers** leads to two parallel subgoals: **Mask-Observe** performed by the Escort subteam and **Pickup** performed by the Transport subteam. The programmer must represent the domain-specific constraint that the Transport subteam cannot perform **Pickup** until the the Escort subteam has reached its masking locations and begun observing. Again, once the programmer has specified the high-level constraint, the TOP infrastructure handles the generation of any communication necessary for the proper synchronization.

## 3 TOP Implementation: TEAMCORE

Our current TEAMCORE system uses distributed TEAMCORE wrapper agents, based on the Soar [17] integrated agent architecture, to support many of the features required by our team-oriented programming framework through. We can categorize each TEAMCORE wrapper agent's teamwork expertise as containing the domain-specific team-oriented program and a reusable domain-independent component.

### 3.1 STEAM Component of TEAMCORE

Our previous work on STEAM [20] provides a significant component of TEAMCORE's general-purpose teamwork model. This model encodes domain-independent rules that explicitly outline team members' responsibilities and commitments in teamwork.
STEAM uses joint intentions theory [14] as the basic building block of teamwork, but SharedPlans theory [9] has also strongly influenced it. STEAM's teamwork knowledge consists of three classes of domain-independent axioms: (i) *Coherence preserving* rules require team members to communicate to ensure coherent initiation and termination of team plans, e.g., a team member must inform others if it uncovers crucial information that would render a team plan unachievable; (ii) *Monitor and repair* rules ensure that team members will substitute for other critical team members who have failed in their roles; (iii) *Selectivity-in-communication* rules use decision theory to weigh communication costs and benefits to avoid excessive communication in the team. STEAM's 300 Soar rules are available in public domain and have been used in several different domains reported in the literature [21].

### 3.2 TEAMCORE Extensions to STEAM

The TEAMCORE system extends the original STEAM model to support the following features desired for full TOP functionality:

- Teamwork knowledge encapsulated within wrapper agents to support heterogeneous domain-level agents
- KQML point-to-point and multicast communication
- Automatic generation of task requests to domain-level agents
- Automatic generation of monitoring requests to domain-level agents
- TOP Interface: a GUI to facilitate specification of organizations and team plans

In the original STEAM implementation, the teamwork knowledge resided directly in the domain-level agent's knowledge base. TEAMCORE places this knowledge in a separate wrapper agent, so that we no longer rely on an ability to modify code in the domain-level agent. The TEAMCORE system supports the use of KQML for all inter-agent communication. The wrapper agents have automatic procedures for sending/receiving the KQML messages appropriate for tasking the domain-level agent in service of the current team goals. Figure 5 represents the system structure of the TEAMCORE implementation of the evacuation scenario. For simplicity, we illustrate only point-to-point communication among TEAMCORE agents, but in reality, they broadcast messages to the other TEAMCORE members of the relevant team. In addition, this illustration depicts only three of the sixteen helicopter agents in the full implementation.

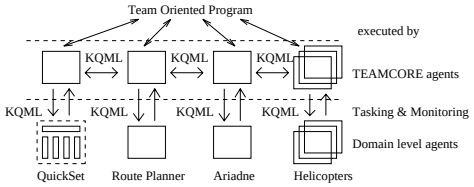


**Fig. 5.** Agent structure and communication in TEAMCORE evacuation scenario implementation.

Although the STEAM rules provide TEAMCORE agents with an automatic procedure for communicating private beliefs amongst themselves, they do not provide a procedure for communicating beliefs between a TEAMCORE agent and its domain-level agent. However, it is the domain-level agent, and not the TEAMCORE agent itself, that has access to most observations relevant to the termination conditions of the team plans (e.g., a pilot agent can observe that another helicopter has crashed). On the other hand, knowledge of the current team plans resides in the TEAMCORE agent, so the domain-level agent may not know what observations are relevant (e.g., a pilot agent may not know which other helicopters are in its team).

We have extended the TEAMCORE agents' domain-independent knowledge to include the generation of appropriate monitoring requests. STEAM already requires models of the conditions under which team goals become achieved, irrelevant, or unachievable. The TEAMCORE agent also has a capability specification describing the conditions that its domain-level agent can observe. From this model of monitoring capabilities, we derive our automatic procedure for generating monitoring requests. The capability specification also specifies how to translate monitoring responses into private beliefs of the TEAMCORE agent, who may ultimately communicate them to other TEAMCORE agents according to the standard STEAM procedures.

We have also created a TOP Interface (TOPI) to facilitate the programmer's effort in specifying the team organization and plans. Figure 6 shows a sample screen shot in programming the evacuation scenario, where the three panes correspond to the plans, organization, and domain-level agents, from left to right. The programmer can specify operators and requirements in the left pane and then assign roles from the organiza-

tion in the middle pane to these operators. The roles and teams in the organization inherit the requirements from their assigned operators, and the programmer can assign domain-level agents from the rightmost pane to these roles. In Figure 6, the programmer has attempted to assign the agent “RPlan” from the rightmost pane to the “Obtain orders” role in the middle pane. TOPI notices the capability mismatch and notifies the programmer with the crossed-out icon. TOPI uses the specifications and assignments to generate a Soar file used by the wrapper agents in running the scenario.

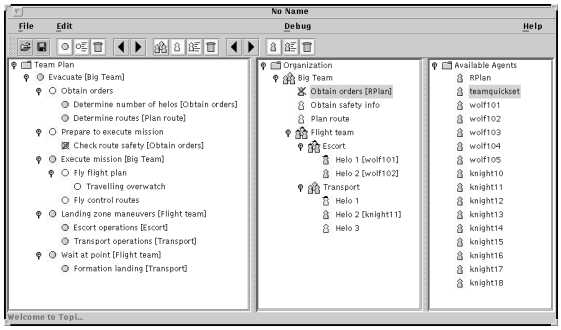


**Fig. 6.** Screenshot of sample organization and operator specification using TOPI.

### 3.3 Evacuation Scenario

In applying TEAMCORE to our evacuation scenario, the team programmer could use the following agents (as in Figure 1):

- **Quickset:** (from P. Cohen et al., OGI) Multimodal input agents [C++, Windows NT]
- **Route planner:** (from Sycara et al., CMU) Path planner for aircraft [C++, Windows NT]
- **Ariadne:** (from Minton et al., USC ISI) Database engine for dynamic threats [Lisp, Unix]
- **Helicopter pilots:** (from Tambe, USC ISI) Agents flying simulated helicopters [Soar, Unix]

The agents provided a fixed specification of possible communication and task capabilities. None of the agents had pre-existing social capabilities, so the TEAMCORE wrapper agents are responsible for all teamwork behavior. The wrapper agents also have knowledge of the domain-specific team plans, containing a hierarchy of 18 joint goals (Figure 4 represents part of this hierarchy).

### 3.4 Evaluation of Evacuation Scenario

The TEAMCORE-based teams have so far successfully met the initial 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 ($18 \times 3\times$ 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.

### 3.5 Research Issues for the Current Implementation

The TEAMCORE system marks a significant stepping stone along the path to our ultimate goal of team-oriented programming. Previous research into the integration of heterogeneous research has focused on the problem of syntactic/semantic interoperability among agents [6]. However, our work in solving the problems addressed by TEAMCORE has unearthed novel challenges in guaranteeing flexible team behavior.

One key issue is the proper generation of monitoring and tasking requests. TEAMCORE's current mechanism automatically generates a monitoring request for all agents capable of monitoring a condition, which can lead to multiple agents monitoring for the same event. Although such redundancy can provide robustness in certain situations, it can also waste agent resources in others. We must explore such issues more fully in order to design a correct mechanism for monitoring and tasking.

The lack of general capability descriptions is another obstacle to the current TEAMCORE system's achievement of full TOP functionality. Once we implement the full capability language described in Section 2, then we could use specifications of plan requirements and agent capabilities to create the team membership and structure at run time. The current TEAMCORE implementation also assumes that there are agents currently available that meet each plan's exact requirements, but it is sometimes the case that no agent meets the exact requirements. However, there might be agents that can perform "capability transformation" to mediate an inexact match. For instance, in

the evacuation scenario, Quickset and the route planner represent points as latitude-longitude pairs, whereas the helicopter pilot agents use a different coordinate system, $x$-$y$-cell. In this case, we changed the pilot code to translate the coordinates, but we may not always have the ability to change the domain-level agents' code. In addition, the original route planner code provided routes for tanks. A route with points less than 10m apart may be suitable for tanks, but it produces undesirably jerky flight patterns in helicopters. In this case, the designers of the route planner modified its behavior to provide fewer intermediate points. However, a more general approach would recognize the mismatch at run time and invoke a translator agent capable of the appropriate mediation.

There are also limitations in the TEAMCORE wrapper agents' representation of their domain-level agents. In the current implementation, the TEAMCOREs autonomously make commitments and assign tasks for the domain-level agents, who may have their own tasks, goals, and preferences to consider before taking on any new commitments for our organization. Many researchers have investigated various methods for allowing agents to balance these competing demands [3, 4, 10, 27]. We could potentially extend our architecture to incorporate such methods in reasoning about what commitments a TEAMCORE can make on behalf of its individual agent.

## 4 Related Work

Many of the issues that need to be addressed in a team-oriented programming environment have been raised in various designs and implementations of teamwork, and have been influential in shaping our ideas. However, as outlined below, TEAMCORE is unique in synthesizing many of these ideas and realizing them via a distributed set of agents that integrate a heterogeneous set of existing agents.

Tidhar [23, 24] used the term "team-oriented programming" to describe a conceptual framework for specifying team behaviors based on mutual beliefs and joint plans, coupled with organizational structures. This framework forms the basis of an implementation based on the dMars agent architecture [25]. In Tidhar's framework, the organizational hierarchy ensures that only appropriate agents (e.g., team leaders) fill specific roles offering certain authority or privilege. Tidhar describes how one can (automatically) unfold team plans into plans for individual agents containing communicative acts that ensure rudimentary coordination. His framework also addressed the issue of team selection [26] — team selection matches the "skills" required for executing a team plan against agents that have those skills.

While many of the features of Tidhar et al.'s conceptual and implemented frameworks are important in the context of TEAMCORE, the critical issue of agent reuse, particularly involving heterogeneous (non-dMars) agents, is not given much attention. As seen in Section 3.2, reusing existing agents by wrapping them requires, at the very least, the addition of tasking and monitoring capabilities, as well as a communication infrastructure, to the teamwork layer. Furthermore, TEAMCORE's flexibility of reorganization and communication selectivity available through STEAM does not seem to be part of the abstract team-layer of SWARMM.

Other implementations of model-based teamwork reasoning relevant to the current work include GRATE* [11] and COLLAGEN [18]. GRATE* implements a model of

cooperation based on the joint intentions framework, similarly used by STEAM. Each agent has its own *cooperation level* module, which handles negotiating involvement in a joint task and maintaining information about its own and other agents' involvement in joint goals. COLLAGEN models dialogue between a user and an agent — a form of joint activity — based on the SharedPlans [9] model of joint action. Both these models of collaboration have been compared to that implemented in STEAM in [20]; in particular, TEAMCORE's STEAM module allows teamwork to a deeper level than the single joint goal/plan allowed in GRATE*, and also provides capabilities for monitoring role performance and role substitution in repairing team activity.

Regarding the specific issue of agent reuse, both GRATE* and COLLAGEN have a fairly clean separation of the teamwork layer from the individual problem-solving layer of an agent. However, these systems have not explicitly focused on team-oriented programming as outlined in this article. COLLAGEN in particular targets wrapping a single agent for collaboration with a human user, so that the issue of programming a team of agents is not particularly relevant. In this context, GRATE* is more similar to the TEAMCORE effort, but the more complex nature of the teams and team tasks in TEAMCORE has led us to explicitly focus on TOP and to explore several novel issues (e.g., automatic generation of monitoring conditions) that GRATE* does not address.

The ADEPT architecture for modeling business processes [12] allows a more flexible, hierarchical team organization than GRATE*. ADEPT consists of multiple *agencies*, each containing a *responsible agent*, which handles communication and interaction with other agencies. Each agency's "capabilities" are maintained by the various responsible agents, avoiding the use of a central facilitator or broker. A task is "contracted out" to an agency which has the capabilities to perform that task. As with GRATE*, ADEPT provides a fairly clean interface between the individual task-achieving agents and the social level. However, the ADEPT framework does not seem to address the issue of agent reuse directly, although the architecture itself could potentially incorporate heterogeneous agents. Also, ADEPT does not provide an explicit model of teamwork, such as that based on joint plans/intentions (the basis of collaboration seems more closely related to Castelfranchi's notion of *social commitment* [5]).

Singh [19] has recently proposed an abstract framework for coordinating heterogeneous agents. Singh's model represents planned activity via finite-state automata (abstracting away the internal workings of the agents), where transitions represent external actions or events. The coordination service maintains knowledge of individual agents' actions as well as the overall joint plan and, upon receiving a request to perform an action, informs the appropriate agents as to whether an intended action should be executed, delayed, or omitted so as to fit with the joint activity of other agents. Singh's model does not address many of the issues of teamwork; however, it provides a potentially useful tool which could augment the joint plan framework of TEAMCORE with a language for specifying flexible, coordinated interactions at an abstract level.

Like the STEAM rule module within TEAMCORE, the COOL coordination framework [1] also focuses on general-purpose coordination by relying on obligations among agents. However, it explicitly rejects the notion of joint goals and joint commitments. It would appear that individual commitments in COOL would be inadequate in address-

ing some teamwork phenomena, but further work is necessary in understanding the relationship between COOL and TEAMCORE.

Code reuse and its automated support are important issues in software engineering. Meyer's notion of *design by contract* [16] involves the use of functional abstractions of software modules (i.e., preconditions and postconditions) to safely allow the incorporation of third-party software, analogous to defining agents' capabilities. The CHAIMS system [2] uses *megaprograms* to perform operations across large heterogeneous multi-site software systems—such megaprograms reuse existing software by wrapping them with a small program that manages their execution and handles communication with the central megaprogram. While such concerns are obviously related to our focus on agent reuse, the components are not assumed to behave autonomously, and tasks and organizations do not change dynamically. Hence, many of the issues of concern to us do not arise. Furthermore, by exploiting notions of teamwork, we are able to provide many of the same coordination and communication services automatically.

## 5 Conclusion and Future Work

Our team-oriented programming (TOP) effort is motivated by the need for rapid development of agent teams from existing, heterogeneous, distributed sets of agents. To this end, we are developing TEAMCORE, a reusable, domain-independent infrastructure to support TOP. The TEAMCORE wrapper agents form a distributed team-readiness layer for augmenting domain-level agents with the following social capabilities:

- Coherent commitment and termination of joint goals
- Team reorganization in response to member failure
- Selective communication
- Incorporation of heterogeneous agents
- KQML communication infrastructure
- Automatic generation of tasking and monitoring requests

We believe that the distributed TEAMCORE agents represent a significant advance toward TOP. Indeed, TEAMCORE greatly simplified our efforts to render agents team-ready and enable them to function coherently towards the joint goal of evacuation. We completely reused the social capabilities listed above, so that our only remaining task was to specify the team program. We successfully generated a correct team program using the goal hierarchy of Figure 4 and the organization hierarchy of Figure 3.

Our initial success in developing a team-oriented program for the evacuation scenario and its (at least current) ease of modification indicates the utility of our TOP framework. More rigorous experiments are clearly necessary to validate these claims. By comparing the programming effort needed to construct teams with TEAMCORE against the effort required when using other techniques, we hope to provide some quantification of TEAMCORE's benefit. We could also quantify the robustness of the organizations constructed with TEAMCORE by evaluating the system performance under various numbers and types of failures during the execution.

The implementation of the TOP framework via TEAMCORE wrapper agents has identified several key research issues, as Section 3.5 illustrates. However, there are

other novel issues on our agenda as well. In our current implementation, there is a one TEAMCORE wrapper for each domain-level agent. However, a single TEAMCORE agent could potentially wrap multiple domain-level agents, reducing team communication. However, the increased centralization could cause greater computational loads, while also rendering failure of a TEAMCORE agent more catastrophic. A more thorough analysis of these tradeoffs should support an automatic procedure for generating optimal structures of TEAMCORE agents.

We are also investigating the use of machine learning in TEAMCORE. In particular, learning from team failures would enable TEAMCORE agents to correct missing (or incorrectly specified) coordination constraints, or modify the existing organization hierarchy to more appropriately match the task at hand. Again, the key is that this learning improves the team-level interactions, rather than the skills of the individuals.

## Acknowledgments

This research was supported by DARPA award No F30602-98-2-0108, under the Control of Agent Based Systems program. The effort is being managed by ARFL/Rome Research Site. We thank Phil Cohen, Katia Sycara and Steve Minton for collaboration on the TEAMCORE project, and for providing the Quickset, route-planner and Ariadne Web-based query agents respectively. We also gratefully acknowledge the support of Hank Seebeck of GlobalInfotek for hosting the different agent-systems mentioned in this paper, and ensuring their smooth integration via TEAMCORE.

## References

1. Mihai Barbuceanu and Mark Fox. The architecture of an agent building shell. In M. Wooldridge, J. Muller, and M. Tambe, editors, *Intelligent Agents, Volume II: Lecture Notes in Artificial Intelligence 1037*. Springer-Verlag, Heidelberg, Germany, 1996.
2. Dorothea Beringer, Catherine Tornabene, Pankaj Jain, and Gio Wiederhold. A language and system for composing autonomous, heterogeneous and distributed megamodules. In *Proc. of the DEXA International Workshop on Large-Scale Software Composition*, 1998.
3. Guido Boella, Rossana Damiano, and Leonardo Lesmo. Cooperating to the group's utility. In N.R. Jennings and Y. Lespérance, editors, *Intelligent Agents VI — Proc. of the Sixth International Workshop on Agent Theories, Architectures, and Languages (ATAL-99)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2000. In this volume.
4. Sviatoslav Brainov. The role and the impact of preferences on multiagent interaction. In N.R. Jennings and Y. Lespérance, editors, *Intelligent Agents VI — Proc. of the Sixth International Workshop on Agent Theories, Architectures, and Languages (ATAL-99)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2000. In this volume.
5. Cristiano Castelfranchi. Commitments: from individual intentions to groups and organizations. In *Proc. of International Conference on Multi-Agent Systems*, pages 41–48, 1995.
6. Paul Cohen, Robert Schrag, Eric Jones, Adam Pease, Albert Lin, Barbara Starr, David Gunning, and Murray Burke. The DARPA high-performance knowledge bases project. *AI Magazine*, 19(4):25–49, 1998.
7. Philip R. Cohen, Michael Johnston, David McGee, Sharon Oviatt, Jay Pittman, Ira Smith, Liang Chen, and Josh Clow. Quickset: Multimodal interaction for distributed applications. In *Proc. of the Fifth Annual International Multimodal Conference*, pages 31–40, 1997.

8. James Firby. An investigation into reactive planning in complex domains. In *Proc. of the National Conference on Artificial Intelligence*, 1987.
9. Barbara Grosz and Sarit Kraus. Collaborative plans for complex group actions. *Artificial Intelligence*, 86:269–358, 1996.
10. Lisa Hogg and Nick Jennings. Variable sociability in agent-based decision making. In N.R. Jennings and Y. Lespérance, editors, *Intelligent Agents VI — Proc. of the Sixth International Workshop on Agent Theories, Architectures, and Languages (ATAL-99)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2000. In this volume.
11. Nick Jennings. Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. *Artificial Intelligence*, 75, 1995.
12. Nick Jennings, T. J. Norman, and P. Faratin. ADEPT: An agent-based approach to business process management. *ACM SIGMOD Record*, 27(4):32–39, 1998.
13. Craig A. Knoblock, Steven Minton, Jose Luis Ambite, Naveen Ashish, Pragnesh Jay Modi, Ion Muslea, Andrew G. Philpot, and Sheila Tejada. Modeling Web sources for information integration. In *Proc. of the National Conference on Artificial Intelligence*, 1998.
14. Hector J. Levesque, Philip R. Cohen, and José Nunes. On acting together. In *Proc. of the National Conference on Artificial Intelligence*. Menlo Park, Calif.: AAAI press, 1990.
15. 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.
16. Bertrand Meyer. Applying design by contract. *Computer (IEEE)*, 25(10), 1992.
17. Allen Newell. *Unified Theories of Cognition*. Harvard Univ. Press, Cambridge, MA, 1990.
18. Charles Rich and Candace Sidner. COLLAGEN: When agents collaborate with people. In *Proc. of the International Conference on Autonomous Agents (Agents'97)*, 1997.
19. Munindar P. Singh. A customizable coordination service for autonomous agents. In *Proc. of the Fourth International workshop on Agent Theories, Architectures and Languages (ATAL'97)*, 1997.
20. Milind Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, 7:83–124, 1997.
21. Milind Tambe, Jafar Adibi, Yaser Alonaizon, Ali Erdem, Gal Kaminka, Stacy Marsella, and Ion Muslea. Building agent teams using an explicit teamwork model and learning. *Artificial Intelligence*, 110(2), 1999.
22. Milind Tambe, W. Lewis Johnson, Randolph Jones, Frank Koss, John E. Laird, Paul S. Rosenbloom, and Karl Schwamb. Intelligent agents for interactive simulation environments. *AI Magazine*, 16(1), Spring 1995.
23. Gil Tidhar. Team-oriented programming: Preliminary report. Technical Report 41, Australian Artificial Intelligence Institute, 1993.
24. Gil Tidhar. Team-oriented programming: Social structures. Technical Report 47, Australian Artificial Intelligence Institute, 1993.
25. Gil Tidhar, Clint Heinze, and Mario Selvestrel. Flying together: Modelling air mission teams. *Journal of Applied Intelligence*, 8(3), 1998.
26. Gil Tidhar, Anand S. Rao, and Elizabeth A. Sonenberg. Guided team selection. In *Proc. of the Second International Conference on Multi-Agent Systems*, 1996.
27. Thomas Wagner and Victor Lesser. Relating quantified motivations for organizationally situated agents. In N.R. Jennings and Y. Lespérance, editors, *Intelligent Agents VI — Proc. of the Sixth International Workshop on Agent Theories, Architectures, and Languages (ATAL-99)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2000. In this volume.
28. Mike Williamson, Katia Sycara, and Keith Decker. Executing decision-theoretic plans in multi-agent environments. In *Proc. of the AAAI Fall Symposium on Plan Execution: Problems and Issues*, November 1996.