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. Indeed, to succeed in highly uncertain, complex applications, the agent teams must be both robust and flexible. Unfortunately, development of such 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, but abstracts away from the large number of coordination plans potentially necessary to ensure robust and flexible team operation. We support TOP through a distributed, domain-independent teamwork 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, including user interfaces [14], robotic space missions, virtual training environments [17], and information extraction on the Internet [23]. These agents are often autonomous, heterogeneous, and distributed over a variety of platforms and environments. Yet, when novel, complex tasks arise, users may desire such diverse agents to work together to accomplish the task. Such reuse of existing agents is preferable to the building of 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, 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

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

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 (there could be multiple realizations of our view of TOP). 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) [11], 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. Furthermore, by suitable substitution, we wish to be able to quickly reconfigure the team for other tasks (e.g., to monitor enemy activity on a battlefield).

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) [4], a route-planning agent, and an information-gathering agent (Ariadne) [9]. 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 these agents communicating over the Internet.

2 Team-Oriented Programming

In our general framework, the team-oriented programmer must develop a system where agents act as a team in performing a joint task. The programmer has a pool of agents available — such agents may be developed by different designers, based on different underlying 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.

2.1 Agent Capability Descriptions

We assume that each agent that is a potential team member has a functional interface describing its *capabilities*. An agent's set of capabilities is an abstraction, so there are no details on how an agent would go about performing a task. The capability description for an agent may specify: (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.

```
(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 has the capability to perform the **Plan-Route** task, where the task request message must provide the start and destination points as input parameters. There is a constraint that the points, restricted to a particular geographical window, must be specified in terms of latitude-longitude coordinates. The output is a planned route, in terms of a list of points, again specified in latitudes and longitudes. In contrast, a helicopter pilot agent can perform several tasks. One of the tasks is coordinate 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. There is no output in this case. The helicopter pilot agent can also perform the task of monitoring, by observing features of its own vehicle, the terrain, or other agents in the environment. For a helicopter pilot agent to monitor the status of another helicopter, the parameters must identify the specific vehicle in the environment, the attribute to be monitored (status), and the request for notification when there is a change in status. The output is either that the specified vehicle's status is "normal" or "destroyed".

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. The tasks may not necessarily provide outputs. 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 an earlier task sent to it. 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 as discussed above.

While the capability description above 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.

2.2 The Team-Oriented Program

Given a pool of agents and descriptions of their capabilities, the team-oriented programmer's job is to implement only the problem-specific aspects of the system. We believe that 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

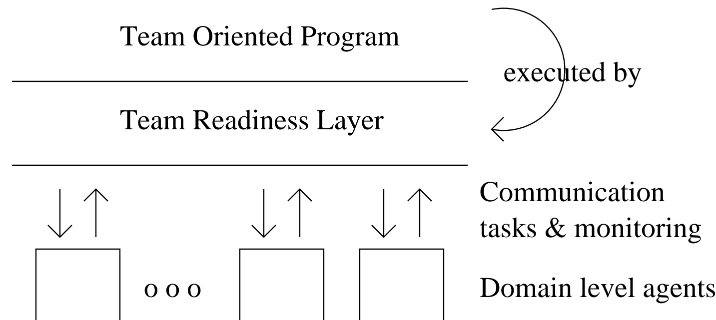

Fig. 2. An abstract view of team-oriented programming.

This team-oriented program is to be executed via a TOP infrastructure, which consists of a "team-readiness" layer. Figure 2 illustrates our view of the TOP infrastructure that enforces team behavior. The "team-readiness" layer provides an interface for agents written in different languages. It ensures coherent execution of team plans by providing task instructions to the domain-level agents at the appropriate times (where the task instructions may initiate tasks, terminate tasks already requested, or specify attributes to monitor). This team-readiness layer hides the details of the coordination behavior, low-level tasking, and flexible re-planning, allowing the programmer to consider only the details at the level of abstraction of the team specification. 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).

In the team-oriented program, the organization hierarchy is specified using roles that may be filled by individual agents or groups of individual agents. For an example of such a hierarchy,

we continue with our example of the evacuation scenario. Figure 3 illustrates a portion of the organization hierarchy of the agents involved with this scenario. Each leaf node corresponds to an individual role, while the internal nodes correspond to subteams of these 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 [5], 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. Having common knowledge of team procedures is akin to providing the team with the knowledge of "standard operating procedures" in a military setting.

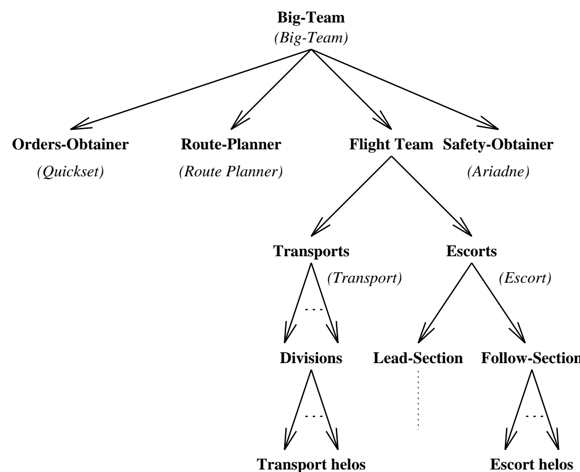

Fig. 3. Partial organizational hierarchy of agents in evacuation team.

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. Once the programmer specifies the initiation conditions, the TOP infrastructure ensures that the team will synchronize itself appropriately in executing the plan. Thus, the programmer need not program such synchronization actions. For instance, there is no need to specify how the members of the top-level team, Big-Team, must synchronize themselves to start jointly achieving **Process-Orders**. 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 must also explicitly specify the termination conditions, the conditions under which a reactive team plan is achieved, irrelevant or unachievable. Such explicit specification within the team program ensures that *all* team members have access to this knowledge, so that the team can terminate the goal coherently. Once again, the programmer does not have to specify the procedure for coherent reactive plan termination. Instead, the TOP architecture can use the conditions of achievement, irrelevance, and unachievability as the basis for automatically generating the communication necessary to jointly terminate a team plan. These explicit termination

conditions also support the automatic generation of monitoring requests. If certain agents have the capability to observe conditions that reflect any termination condition of a team goal, then the TOP architecture can automatically task them to report on any change in that condition.

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 has the option of specifying particular agents or subteams, taken from the organization hierarchy, to perform a given goal. On the other hand, the team programmer may specify the performing agents using only abstract role labels (as in Figure 4), in which case the TOP infrastructure assigns agents/subteams from the organization hierarchy 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. In this particular program, the agents of Big-Team jointly perform the top-level goal of **Evacuate**.

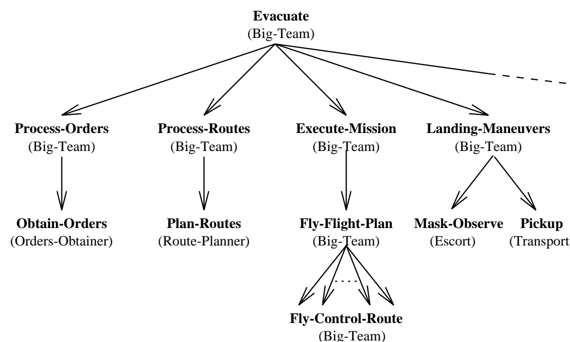

Fig. 4. Partial reactive plan hierarchy for evacuation scenario.

With respect to the actions of the reactive team plans, the high-level team-plans, such as **Evacuate** in Figure 4, typically have few actions. Instead, these plans decompose into a sequence of subgoals. For instance, **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 will generate the appropriate tasking messages at run-time by examining the current parameters of the **Obtain-Orders** goal and the capability specification of the current Orders-Obtainer.

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 supports many of the features required by our Team-Oriented Programming framework. The team-readiness layer in the implemented system consists of distributed TEAMCORE wrapper agents based on the Soar [13] integrated agent architecture. We can categorize each TEAMCORE wrapper agent's teamwork expertise as containing a domain-specific and a domain-independent component, the latter forming the central part of TEAMCORE. The domain-specific part consists of the organization hierarchy and the reactive team plans, i.e., *team operators* in Soar.

Common knowledge of team operators does not protect the team members from incomplete or incorrect beliefs, from unexpected failures and successes, from unexpected communication failures, and so on. The domain-independent component of TEAMCORE must surmount such uncertainties of dynamic complex domains and adapt to the current environment. The heart of the domain-independent component is teamwork reasoning based on a general model of teamwork.

3.1 STEAM Component of TEAMCORE

Our previous work on STEAM [18] provides a significant component of TEAMCORE's general-purpose teamwork model. This model encodes domain-independent axioms and theorems that explicitly outline team members' responsibilities and commitments in teamwork. In essence, it attempts to provide agents with the common- knowledge of teamwork that would enable them to autonomously reason about coordination and communication. STEAM uses joint intentions theory [10] as the basic building block of teamwork, but SharedPlans theory [6] has also strongly influenced it, as have the constraints of real-world application domains. STEAM builds on our experience in building agent teams for a variety of domains [16]. When working to achieve a team operator, agents bring to bear all of the reasoning power of the general STEAM teamwork model, facilitating their communication and coordination.

Currently, we can divide STEAM's teamwork knowledge into three classes of domain-independent axioms. The first class consists of *coherence preserving* (CP) axioms that require team members to communicate with others to ensure coherent initiation and termination of team goals. For instance, if an agent privately discovers that the currently selected team goal is unachievable, then it must not unilaterally terminate the goal. 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 goal. As an example of STEAM's application, consider the helicopter team flying in formation. If one of the helicopter pilots observes unanticipated enemy vehicles on their flight route, STEAM's CP axioms require that this pilot inform its teammates about the enemy units (since it makes the relevant team operator unachievable), so that the team reacts coherently to the enemy.

Monitor and repair (MR) axioms detect whether a team task is unachievable due to unexpected member (individual or subteam) failure. When such a failure occurs, the axioms then lead the team into reorganization — via reassignment of roles — to overcome this failure. For example, if a helicopter crashes, then the MR axioms would allow the team members to observe that their current flight goal is unachievable. The axioms would also trigger the reorganization of the flight team, perhaps also changing low-level formation plans, to allow the team to continue pursuing its overall mission.

STEAM requires *selectivity-in-communication* (SC) axioms, 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. The key idea is to explicitly reason about the costs and benefits of different techniques for attaining mutual beliefs. For instance, in some cases, if there is a high likelihood that an agent's teammates can obtain the relevant information via their own observation (even if with a slight delay), then it should avoid costly communication. In contrast, communication becomes essential

if there is a low likelihood that teammates can obtain the relevant information independently, and if there is a high cost to not making this information available to the teammates.

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 the specification of the organization 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. We do require that the domain-level agent provide a communication interface, so that the wrapper agent can submit the necessary task requests. 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 to the current team goals. Figure 5 represents the agent and communication structure of the TEAMCORE implementation, using an example from the evacuation domain. For simplicity, we illustrate only point-to-point communication among the TEAMCORE agents, but we have also added multicast capabilities to allow the TEAMCORE agents to broadcast messages to the other TEAMCORE members of a relevant team. In addition, this illustration depicts only three helicopter agents, but the current implementation includes up to *sixteen* such agents.

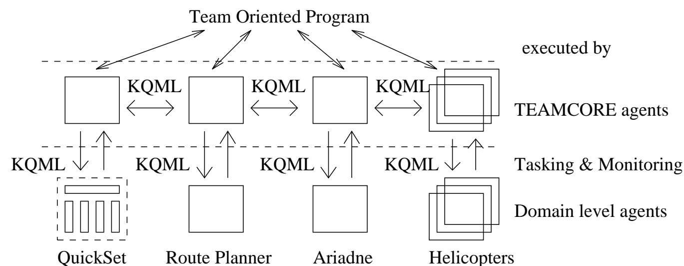

Fig. 5. Agent structure and communication flow in TEAMCORE implementation of evacuation scenario.

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

We have extended the TEAMCORE agents' domain-independent knowledge to include the generation of appropriate monitoring requests that prompt the domain-level agent about which

possible events are currently relevant. For each team goal, STEAM already requires that the TEAMCORE agents maintain models of the conditions under which the goal becomes achieved, irrelevant, or unachievable. The TEAMCORE agent's model of the wrapped domain-level agent includes a capability specification describing the conditions that it can observe, so this model of monitoring capabilities forms the basis for 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 screenshot 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 roles and role requirements in the middle organizational pane, and then assign team-ready (i.e., wrapped) domain-level agents from the rightmost pane to these roles. In Figure 6, the programmer had 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. The user has highlighted the mismatched role and a new agent "teamquickset" as part of the process of making the correct assignment. In addition, one can then assign roles from the organization to operators created in the leftmost pane. TOPI uses the specifications and assignments to generate a Soar file used by the wrapper agents in running the scenario.

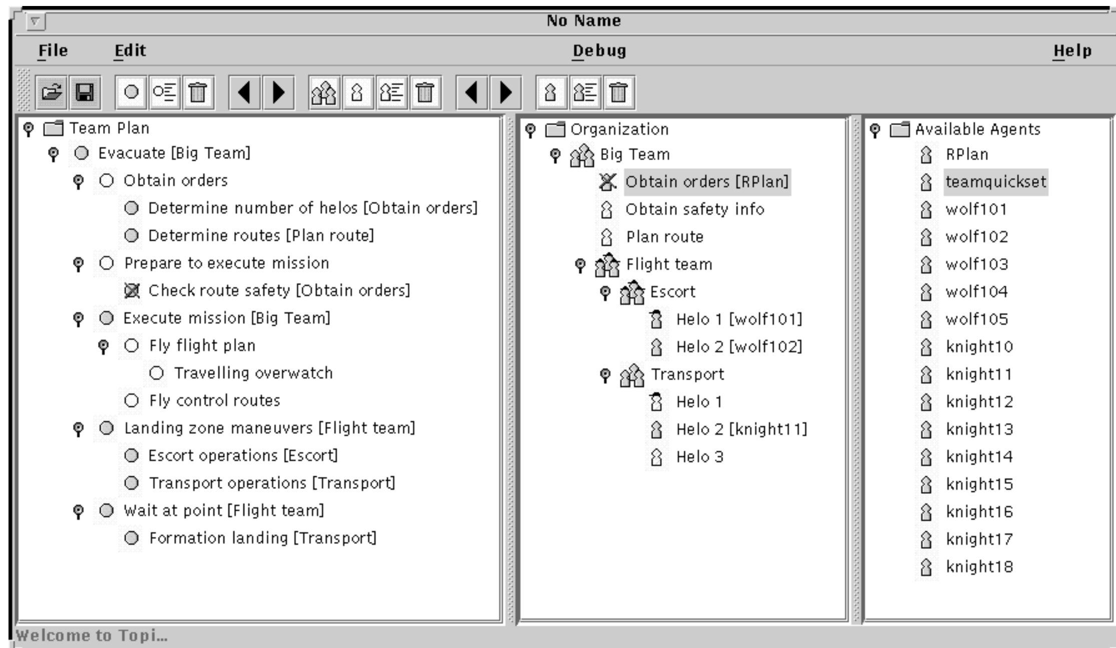

Fig. 6. Screenshot of sample organization and operator specification using TOPI.

3.3 Evacuation Scenario

We have applied TEAMCORE to the problem of evacuating civilians from a threatened location. The team programmer had the following agents (with capabilities as in Figure 1) available:

Quickset: (developed by P. Cohen et al., Oregon Graduate Institute) C++-based agents running on an NT platform (and connected through an OAA blackboard) enable a human operator to use speech and gesture to provide the command to evacuate, as well as the locations of the evacuation site and other meaningful route landmarks.

Route planner: (developed by Sycara et al., Carnegie-Mellon University) A C++-based route planner running on a UNIX platform can take the route landmarks, as well as a map of the region, and produce a more detailed route that avoids known obstacles (e.g., no-fly zones).

Ariadne: (developed by Minton et al., USC Information Sciences Institute) A LISP-based wrapper running on a UNIX platform can query a database on more dynamic obstacles (e.g., enemy missile launchers).

Helicopter pilots: Soar-based pilot agents running on a UNIX platform can fly helicopters (within a ModSAF simulation environment) along a specified route and land at specified destinations. These agents are different from the pilots used with STEAM, which had teamwork capabilities explicitly programmed in. The pilot agents used in the work presented here had no such social capabilities.

We developed only the helicopter agents ourselves, while the other agents provided a fixed specification of possible communication and task capabilities. None of the agents had pre-existing social capabilities, so we created the TEAMCORE wrapper agents are responsible for all of the teamwork behavior. All of the wrapper agents share the same domain-independent knowledge of the TEAMCORE teamwork models. The wrapper agents also have knowledge of the domain-specific team plans, corresponding to the goal hierarchy representing the problem domain, as in Figure 4, as well as the associated functional models.

The domain-independent component of the wrapper agents invokes all of STEAM's capabilities. The domain-specific team program contains a hierarchy of 18 joint goals (Figure 4 represents part of this hierarchy). However, only three of these require explicit communication to form joint commitments; STEAM can exercise selectivity in the explicit commitment for the other goals by assuming common knowledge of the execution sequence. Termination of these goals does require explicit communication, since the agents have differing beliefs (based on different domain-level observations) about achievement, irrelevance, and unachievability.

3.4 Evaluation of Evacuation Scenario

The TEAMCORE wrapper agents must generate the correct task and monitoring requests, since this is what ultimately drives the domain-level agents' behavior to successfully accomplish their tasks, within their given environments, both efficiently and accurately. The TEAMCORE-based teams have so far successfully met the initial challenge in their evacuation domains.

However, we are also interested in the effort involved in encoding and modifying agents' teamwork capabilities — comparing the effort with TEAMCORE against the alternatives. The key alternative is reproducing all of TEAMCORE's capabilities by providing the domain-level agents with special-case coordination plans. However, 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 each language used by 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. That is already a total of 54 (18×3) coordination plans. Furthermore, to reproduce TEAMCORE's selective communication, additional special cases would be necessitated — in the extreme case, each combination of values for communication costs and rewards could require a separate special case operator ($54\times$ total combinations, already more than a hundred). Furthermore, we may require separate operators depending on whether the communication occurs throughout the entire team or only a subteam. Of course, it would appear

that all such special cases could be economized in our initial implementation by discovering generalizations, but TEAMCORE already encodes such generalizations to avoid the many special cases.

The TEAMCORE specification has greatly facilitated modifications to the team as well. For instance, the route planner was the last addition to the team. The pre-existing team would successfully execute straight-line routes in response to the human commander's orders. We wanted to add the route planner to specify routes around potential obstacles. We first constructed a TEAMCORE wrapper agent to make the route planner team-ready. However, there was no additional coding effort necessary to add the general-purpose teamwork model of TEAMCORE, nor the domain-specific team program, since we had previously coded both for the existing wrapper agents. To extend the organizational hierarchy, we simply added the route planner as a member of Big-Team. We then added the **Process-Routes** branch of the goal hierarchy to allow for route planning. 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 [3]. However, our work in solving the problems addressed by TEAMCORE has unearthed novel challenges in guaranteeing flexible team behavior. We feel that these issues are themselves an interesting contribution, so this section describes them in the hope that, having brought these issues to light, we can make further reductions in team programming effort with future versions of TEAMCORE.

One key issue is the proper generation of monitoring and tasking requests from the TEAMCORE layer to the domain-level agents. Our current TEAMCORE implementation includes a partially automated mechanism for generating such requests, but it fails to fully address many of the issues involved in generating correct behavior. For example, the current implementation generates a monitoring request for all agents capable of monitoring a condition, even though this 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, as described in Section 2.1. 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. Of course, this raises a new issue in how best to match agent capabilities to plan requirements.

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 represent points in a different coordinate system, x-y-cell. In this case, we changed the pilot code to translate latitude-longitude coordinates into the x-y-cell format, but we may not always have the ability to change the domain-level agents' code. In addition, mismatches may require more complex translations. For instance, the original route planner code provided routes for tanks, which require planning at a much finer granularity than helicopters. 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, in general, we cannot correct all of the possible mismatches that may occur. A more flexible approach would recognize the mismatch at run time and invoke a translator agent capable of the appropriate mediation.

There are also limitations in the flexibility of the TEAMCORE wrapper agents in dealing with the domain-level agents. In our evacuation scenario, all of the domain-level agents had the same lack of social capabilities, so we could re-use the same coordination code in all of the wrapper agents. In general, we may have to tailor the wrapper agents to account for the domain-level agent's social capabilities. For example, we may need to reduce TEAMCORE communication if the domain-level agent can perform some of the communication itself. We could potentially include an agent's social capabilities within its overall capability specification.

4 Related Work

Many of the issues that need to be addressed in a Team-Oriented Programming environment, as we have discussed it, 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 [21, 22] has 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 [19]. A joint plan is executed by a team, consisting of potentially more than one agent, each of which is committed to the achievement of the joint goal associated with that plan. 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 (i.e., automatically deciding on an appropriate collection of agents to act as a team for performing a given task) [20] — team selection matches the "skills" required for executing a team plan (i.e., actions and goals to be performed) against agents that have those skills (i.e., contain plans that address those actions and goals).

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 (such as KQML in TEAMCORE), to the teamwork layer. Furthermore, the flexibility of team reorganization on failure and decision-theoretic communication selectivity available through the STEAM component of TEAMCORE does not seem to be part of the abstract team-layer of SWARMM — i.e., the programmer must supply such reasoning about the team itself, whereas TEAMCORE already provides it through the infrastructure.

Other implementations of model-based teamwork reasoning relevant to the current work include Jennings et al.'s work on GRATE* [8] and Rich and Sidner's COLLAGEN [14]. 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 the process of 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 [6] model of joint action. Both these models of collaboration have been compared to that implemented in STEAM in [18]; in particular, STEAM 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 ac-

tivity, which is not available from other systems. The TEAMCORE system has inherited these features of teamwork by incorporating STEAM into its teamwork model.

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—e.g., in GRATE*, the addition of the cooperation and communication layers to a problem-solving agent could be seen as analogous to making the agent "team-ready" (in the way we have used it above). 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 (such as the automatic generation of monitoring conditions) that are not addressed in GRATE*.

More recent work by Jennings and his coworkers has led to the ADEPT architecture for modeling business processes [7]. ADEPT allows a more flexible, hierarchical team organization. ADEPT consists of multiple *agencies*, each containing a *responsible agent*, which handles communication and interaction with other agencies (via their corresponding responsible agencies), along with a number of sub-agencies. The "capabilities" (i.e., services offered) of each agency are maintained by the various responsible agencies, avoiding the use of a central facilitator or broker. A task is "contracted out" (after a negotiation period) to an agency which has the capabilities to perform that task. The resulting "social contract" may involve parameters such as the maximum allowable duration before a result is required. 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 allows for the potential of incorporating heterogeneous agents, it does not seem to be a specific concern of the project. 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* [2]).

Singh [15] 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 be used to augment the joint plan framework of TEAMCORE with a language for specifying flexible, coordinated interactions at an abstract level. Another key point of contrast seems to be that TEAMCORE is an implemented system, with the teamwork layer realized via a distributed set of TEAMCORE agents.

Code reuse is, of course, an important issue in software engineering. Such reuse is being facilitated on an increasingly larger scale, with automated support for the process. Meyer's notion of *design by contract* [12] involves the use of a functional abstraction of software modules (in the form of preconditions and postconditions specified in a common language) to safely allow the incorporation of third-party software—such an abstraction can be seen as analogous to defining agents' capabilities. The CHAIMS system [1] 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, which uses those modules for computation by remotely invoking their execution. 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, the availability of these agents greatly simplified our efforts to render the various evacuation 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, including possible comparisons with teams developed using other techniques in terms of programming effort.

The actual implementation of the TOP framework via the TEAMCORE wrapper agents has also led to identification of several key research issues. Some of these issues have been identified in Section 3.5. There are however other novel issues on our agenda as well. One key issue we are investigating involves greater flexibility in the structure of the wrapper agents in the team-readiness layer. In our current implementation, there is a one-to-one correspondence between TEAMCORE wrappers and domain-level agents. However, there is no reason why a single TEAMCORE agent could not wrap more than one domain-level agents, since it can easily maintain multiple goal structures. Such consolidation reduces team communication, since a single agent would now commit, modify, and terminate joint goals on behalf of multiple domain-level agents. However, the increased centralization can 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.

Another key area of investigation is the role of machine learning in TEAMCORE. In particular, learning from team failures will enable TEAMCORE agents to correct any 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 occurs at the team-level interactions, rather than on improving 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. Dorothea Beringer, Catherine Tornabene, Pankaj Jain, and Gio Wiederhold. A language and system for composing autonomous, heterogeneous and distributed megamodules. In *Proceedings of the DEXA International Workshop on Large-Scale Software Composition*, 1998.
2. C. Castelfranchi. Commitments: from individual intentions to groups and organizations. In *Proceedings of International Conference on Multi-Agent Systems*, pages 41–48, 1995.
3. 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.
4. 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 *Proceedings of the Fifth Annual International Multimodal Conference (Multimedia '97)*, pages 31–40, 1997.
5. J. Firby. An investigation into reactive planning in complex domains. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1987.
6. Barbara Grosz and Sarit Kraus. Collaborative plans for complex group actions. *Artificial Intelligence*, 86:269–358, 1996.
7. N. R. Jennings, T. J. Norman, and P. Faratin. ADEPT: An agent-based approach to business process management. *ACM SIGMOD Record*, 27(4):32–39, 1998.
8. Nick Jennings. Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. *Artificial Intelligence*, 75, 1995.
9. 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 *Proceedings of the National Conference on Artificial Intelligence*, 1998.
10. H. J. Levesque, P. R. Cohen, and J. Nunes. On acting together. In *Proceedings of the National Conference on Artificial Intelligence*. Menlo Park, Calif.: AAAI press, 1990.
11. 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.
12. B. Meyer. Applying design by contract. *Computer (IEEE)*, 25(10), 1992.
13. Allen Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, MA, 1990.
14. C. Rich and C. Sidner. COLLAGEN: When agents collaborate with people. In *Proceedings of the International Conference on Autonomous Agents (Agents'97)*, 1997.
15. M.P. Singh. A customizable coordination service for autonomous agents. In *Proceedings of the 4th international workshop on Agent Theories, Architectures and Languages (ATAL'97)*, 1997.
16. 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*, (to appear), 1999.
17. 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.
18. Milind Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, 7:83–124, 1997.
19. G. Tidhar, C. Heinze, and M. Selvestrel. Flying together: Modelling air mission teams. *Journal of Applied Intelligence*, 8(3), 1998.
20. G. Tidhar, A.S. Rao, and E.A. Sonenberg. Guided team selection. In *Proceedings of the Second International Conference on Multi-Agent Systems*, 1996.
21. Gil Tidhar. Team-oriented programming: Preliminary report. Technical Report 41, Australian Artificial Intelligence Institute, 1993.
22. Gil Tidhar. Team-oriented programming: Social structures. Technical Report 47, Australian Artificial Intelligence Institute, 1993.
23. M. Williamson, K. Sycara, and K. Decker. Executing decision-theoretic plans in multi-agent environments. In *Proceedings of the AAAI Fall Symposium on Plan Execution: Problems and Issues*, November 1996.