Toward Distributing Autonomy over Robot Teams

Anthony Barrett

Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive, M/S 126-347
Pasadena, CA 91109-8099
anthony.barrett@jpl.nasa.gov

Abstract

This paper presents a way to command a system of systems robustly as a single entity. Instead of modeling each component system in isolation and then manually crafting interaction protocols, this approach starts with a model of the collective population as a single system. By compiling the model into separate elements for each component system and utilizing a teamwork model for coordination, it circumvents the complexities of manually crafting robust interaction protocols. The resulting system is both globally responsive by virtue of a team oriented interaction model and locally responsive by virtue of a distributed approach to planning as well as model-based fault detection, isolation, and recovery.

Introduction

NASA mission concepts for the next few decades typically involve progressively larger teams of tightly coordinated spacecraft in dynamic, partially understood environments. In order to manage the teams, each spacecraft must respond to global coordination anomalies as well as local events. Currently techniques for implementing such teams are extremely difficult. They involve either giving one spacecraft tight global control or giving each spacecraft separate commands with explicit communication actions to coordinate. While both approaches work for two or three simple spacecraft, neither scales well to larger populations or more complex spacecraft. New techniques are needed to closely coordinate three or more complex spacecraft.

Flexible teamwork (Tambe 1997) is a technique developed in the multi-agent community for teams of agents that achieve joint goals in the face of uncertainties arising from complex dynamic domains that obstruct coherent teamwork. Flexible teamwork involves giving the agents a shared team plan and a general model of teamwork. Agents exploit this model and team plan to autonomously reason about coordination and communication, providing the requisite flexibility. While this framework has been implemented in the context of real-world synthetic environments like robotic soccer and helicopter-combat simulation, these systems take an ad hoc rule-based approach toward failure diagnosis and response.

Our system takes a model-based approach toward representing teams and their group activities. As Figure 1 shows, a user models a team as a conceptual single entity, but the system distributes the model across the team to move all reasoning as close as possible to the components being reasoned about. The result is a team with elements that are both locally and globally responsive, without a user explicitly thinking about distribution and communication. Planning and multi-agent role assignment are also based on a single model that naturally distributes across multiple coordinated planners.

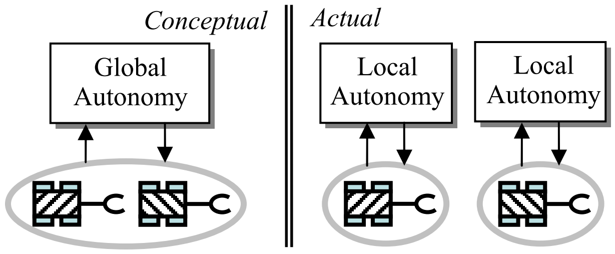

Figure 1. User defines coordinated activities assuming a global executive, and the system handles the distribution and coordination to implement on local executives.

Defining the approach starts with an explanation of the system architecture has three components residing on each robot and how the components interact. Subsequent sections then discuss each of the three components: a feedback controller, and executive, and a planner. Finally the last two sections discuss related work and conclusions.

System Architecture

The system's architecture is a generalization of the three layer approach (Gat 1997) toward defining an agent. The three components in this approach are a feedback controller, a reactive plan executer, and a planner. The feedback controller merges a set of stimulus/response behaviors into feedback loops. As such this layer does not perform any high level reasoning, and can satisfy stringent real-time requirements. The reactive plan executor determines which behaviors are active at any given moment. As such it reasons about system state and how behaviors merge to reach target states. This component still has hard real-time requirements, but they are less demanding than the feedback controller. Finally, the planner reasons about which activities to execute at any given moment. Since such reasoning can get computationally intensive, the planner does not provide hard real-time guarantees.

In our distributed system we place all three components on each robot. While there is a single model of the entire team, the models get broken down into fragments that are distributed to each component, and the components communicate amongst them selves for coordination at various levels. The controllers can tightly control multi-rover activities like rappelling over a cliff, the executives can determine local state and coordinated execution information to robustly manipulate local and team activities, and the planners can coordinate the planning of shared activities to be performed by the group.

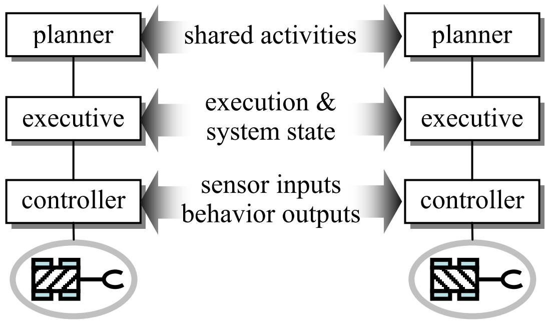

Figure 2. Autonomy is implemented using three identical components that share information at each level.

This work is similar to that of Simmons *et al*. (2002) in that it contains the same components that are organized in the same way. The main difference between our approaches stems from an attempt to use a single model of the collection of robots and then have a compiler automatically distribute the model. Thus our system from a modeler's perspective looks like a single planner-executive-controller collection that manages all robots.

Feedback Controller

A feedback controller implements one or more feedback control loops with varying latency requirements. For fast mobile robots the hard real-time latency requirements are quite stringent to keep robots from running into each other or other obstacles. These feedback loops are typically grouped to implement simple behaviors like drive to a location, turn, and avoid collisions. At any given time some subset of the available behaviors are active, depending on the robot's current activities. For instance, a rover traveling to an object would have the drive to location and avoid obstacles behaviors active. At any given moment two active behaviors might make conflicting demands on a given actuator, so behavior coordination mechanisms are associated with each actuator. For instance, driving might conflict with avoiding obstacles if an obstacle is between a robot and its target location.

The Control Architecture for Multi-robot Planetary OUTpost (CAMPOUT) (Pirjanian *et al*. 2001) distributes such a feedback controller across multiple robots. As illustrated in figure 3, the feedback controllers on each robot host a set of behaviors and behaviors coordinators, but which behavior, or coordinator, belongs on which robot need not be specified. It is quite possible to put all behaviors on one robot for a form of master/slave control, and it is also possible to automatically distribute the behaviors based on latency, communications, and computation requirements.

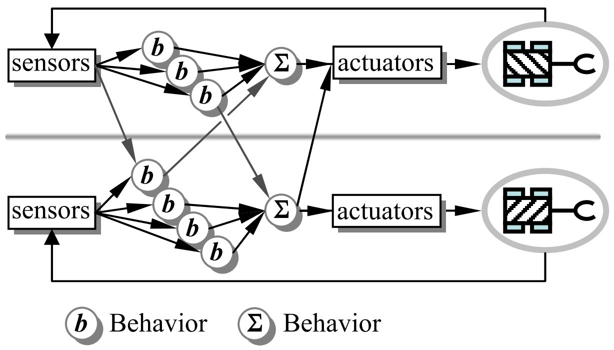

Figure 3. CAMPOUT's feedback controllers coordinate sets of behaviors that can subscribe to information from local and remote sensors as well as provide information for local and remote coordinators.

For instance, suppose that a given pair of robots had two arms, and each arm had respective behaviors to raise and lower a gripper. All that needs to be specified are the 4 behaviors to raise and lower each arm. If one arm happens to be on each robot, then two behaviors will naturally end up on each robot's controller, but the behaviors might end up on one robot if they require a lot of computation and only one robot has a fast processor.

Executive

Once sensor placement, computation and feedback latency requirements have determined the distribution of behaviors across multiple robots, this distribution subsequently determines what each robot's executive reasons about. In general, an executive only reasons about those sensors that it has direct access to and those behaviors that it can directly manipulate.

Each executive has three distinct components: mode identifier, mode reconfigurer, and team sequencer. As illustrated in Figure 4, the mode identifier combines sensory information from the hardware with past commands from the reconfigurer to estimate the system's state. The mode reconfigurer then takes the current state estimate with a target mode from the team sequencer to determine the next commands to pass to a robot's hardware drivers. Finally, the team sequencer procedurally controls the identifier-reconfigurer-driver feedback loop by providing target states to the reconfigurer. At all times local mode identifiers maintain state knowledge, and sequencers react to this information by changing the target state.

While this architecture has been explored for single agent systems using TITAN (Ingham *et al*. 2001), it has yet to be cleanly extended to tightly coordinated teams. This work makes the extension by developing techniques for distributed mode identification, distributed mode reconfiguration, and team sequencing. While these techniques extend the sequencer, they replace the mode identification and reconfiguration components to support distributed computation and hard real-time performance guarantees.

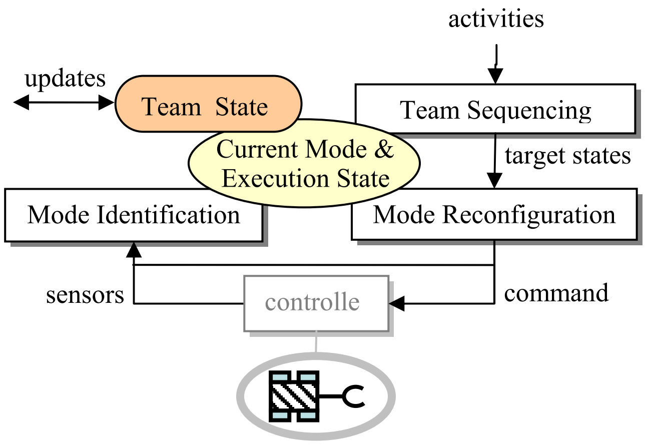

Figure 4. Component architecture of executive on each robotic element, where team state information is kept consistent across all elements.

Distributed Mode Management

By building off of the model-based description language developed for DS-1's Mode Identification & Recovery (MIR) executive (Williams&Nayak 1997), we acquire a representation for explicitly defining the interrelationships between an agent team's complete set of software and hardware components. This facilitates reasoning about how one component's status affects the others' and ultimately sensor observations, which enables taking a set of observations and inferring the team's status. While there are a number of constructs in the language, they all support defining a network of typed components. These components are defined in terms of how a component's modes constraint its port variables, and these constraints are encoded using *variable logic equations* – Boolean equations where the literals are variable equalities.

For instance, the following defines an extremely simplistic system with two components for its robot arms. In this system each arm command refers to a distinct behavior. In this case an arm can be commanded to raise and hold itself at the top or lower and hold itself at the bottom. These behaviors are terminated to stop the arm whenever some stress threshold is exceeded. The arm will stay stopped until the sensed stress falls back below the threshold, at which point the raise or lower behaviors can be restarted

```
(defvalues ArmCmd (raise lower none))
(defvalues bool (T F))

(defcomponent Arm
  :ports ((ArmCmd cmd)(bool stress))
  :modes
((stalled)(stopped)(rising)(falling))
  :transitions
  ((* -> rising  (:and (= stress F)
                       (= cmd raise)))
   (* -> falling (:and (= stress F)
                       (= cmd lower)))
   (* -> stalled (= stress T))
   (stalled -> stopped (= stress F))))

(defsystem rovers
  :sensors
  ((bool LArmAtTop)(bool LArmStress)
   (bool RArmAtTop)(bool RArmStress))
  :affectors ((ArmCmd LArmCmd)
              (ArmCmd RArmCmd))
  :structure
  ((Arm RArm (RArmCmd RArmStress))
   (Arm LArm (LArmCmd LArmStress))))
```

Current model-based diagnosis techniques use some variant of truth maintenance (Nayak&Williams 1997), where components are translated into Boolean equations. These techniques require collecting all observations into a central place and then invoking heuristic algorithms to find the most probable mode that agrees with the observations. While some work has been done to distribute these systems, their underlying algorithms cannot support hard real-time guarantees by virtue of having to solve an NP-Complete problem for each collection of observations. While heuristics can make these algorithms fast on average, the point is that they cannot guarantee performance in all cases.

Instead of working with distributed truth maintenance systems, we will take a simpler approach suggested by knowledge compilation research. This approach involves moving as much of the computation into an off board compilation phase as possible to simplify the onboard computation. Where previous systems take linear time to compile a model and then possibly exponential time to use the compilation to perform mode estimation, our approach takes possibly exponential time to compile a model into a decomposable negation normal form representation and then linear time to perform mode estimation with the equation.

Definition 1: A variable logic equation is in <u>Decomposable Negation Normal Form</u> (DNNF) if (1) it contains no negations and (2) the subexpressions under each conjunct refer to disjoint sets of variables.

For instance, the two robot arm example compiles into a tree-like structure in Figure 5, where the branch that is local to the left arm is a mirror image to that for the right arm.

Given that conjuncts have a disjoint branch variables property, the minimal cost of a satisfying variable assignment is just the cost of a variable assignment with single assignment equations, the minimum of the subexpression costs for a disjunct, and the sum of the subexpression costs for a conjunct. With this observation, finding the optimal satisfying variable assignments becomes a simple three-step process:

1. associate costs with variable assignments in leaves;
2. propagate node costs up through the tree by either assigning the min or sum of the descendents' costs to an OR or AND node respectively; and
3. if the root's cost is infinity or some other value then respectively return failure or descend from the root to determine and return the variable assignments that contribute to its cost.

For instance, Figure 5 illustrates the process of determining that the right arm is stopped. First, observing that RArmStress is false results in assigning values to the RArmStress leaves, and the mode and command leaves get costs related to the last known modes and commands. Second, costs are propagated to the tree root. Third, the root node's cost is used to drill down to find the lowest cost tree with the mode assignment (i.e. RAM_1=stopped).

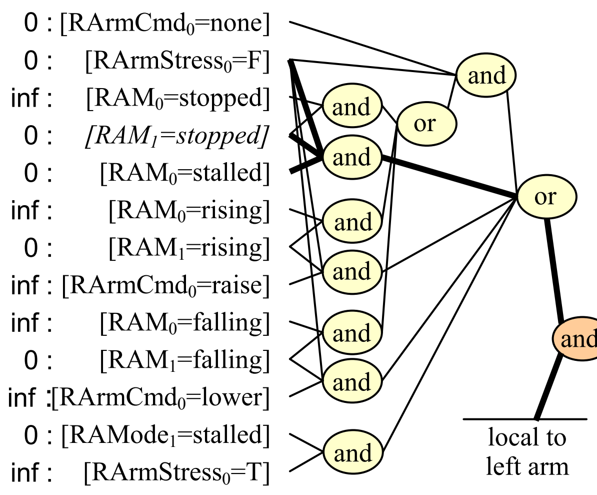

Figure 5. Computing the status of the right arm given past sensor readings and commands is a matter of assigning leaf values and then performing a MINSAT computation.

Finally, it turns out that that the generated DNNF can also be used to compute the commands to reach a target state given the currently estimated state (Barrett 2005). This is because reconfiguration planning can also be couched as a MINSAT problem and the only difference in the planning and diagnosis algorithms involves how to assign leaf costs. Where diagnosis assigns 0 and inf to initial state and command/observation leaves to determine the most likely final state, planning assigns 0 and inf to initial and final state leaves to determine the least cost commands to achieve the final state.

As Figure 5 suggests, the right arm's executive only gets a the piece of the DNNF to reason about the right arm for diagnosis and reconfiguration planning. The distribution of the DNNF starts with the sensors and behaviors that an executive can see and control. These point to leaves of a DNNF that are local to the executive. At this point internal nodes are assigned to a local executive when they only point to leaves or parents assigned to that executive. All other nodes are assigned to the team state. In the two arm example the assignment resulted the rightmost "and" being placed in the team state.

Procedural Control

The distributed onboard sequencer is based both on the Reactive Model-based Programming Language (RMPL) (Williams *et al.* 2001) and a model of flexible teamwork (Tambe 1997) developed within the distributed artificial intelligence community. Flexible teamwork is more than a union of agents' simultaneous execution of individual plans, even if such plans have explicit coordination actions. Uncertainties often obstruct pre-planned coordination, resulting in corresponding breakdowns. Flexible teamwork involves giving agents a shared team plan and a general model of teamwork. Agents then exploit this model and plan to autonomously handle coordination and communication, providing the flexibility needed to overcome emerging unexpected interactions caused either by slight timing delays or anomalies.

RMPL raises the level at which a control programmer thinks about robotic systems. Instead of reasoning about actuators, sensors, and hardware, a RMPL programmer thinks in terms of controlling a system through a sequence of configurations. Thus a control program is written at a higher level of abstraction, by asserting and checking states which may not be directly controllable or observable.

As an example of the rich types of behavior that an RMPL control programmer can encode, the control program below shows a simplistic approach toward defining an activity that gets two rovers to jointly lift a bar. It performs the task by commanding the robot arms into a rising mode whenever they are stopped, and then stopping when one of the robots senses that its arm is at the top position. While only partially shown in this simplistic example, RMPL code can express numerous types of behavior including iteration, conditional branching, concurrent tasks, and preemption.

```
(defactivity liftBar ()
   (do (parallel
          (whenever (= RArm.Mode stopped)
             donext (= RArm.Mode rising))
          (whenever (= LArm.Mode stopped)
             donext (= LArm.Mode rising)))
      watching (:or (= LArmAtTop T)
                    (= RArmAtTop T))))
```

The formal semantics of RMPL has been defined in terms of Hierarchical Constraint Automata (HCA), where the nesting of automata directly corresponds to the nesting of RMPL constructs. For instance, Figure 6 has the HCA for our example, where the outer and two inner boxes respectively correspond to the `do-watching` and the two `whenever-donext` constructs.

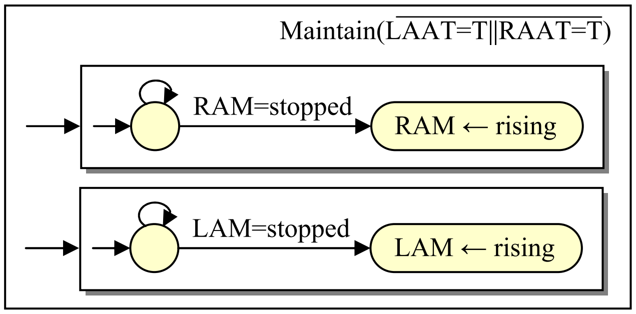

Figure 6. Hierarchical constraint automaton (HCA) for two robot arms lifting a bar, where LAAT, RAAT, LAM, and RAM respectively denote `LArmAtTop`, `RArmAtTop`, `LArm.Mode`, and `RArm.Mode`.

Unlike standard automata, multiple locations in an HCA can be marked. When marked a location stays marked until its target state (if any) has been reached. At which point the mark gets replicated zero or more times over arcs that have true conditions. For instance, the two left locations lack target states, but stay marked by virtue of the loop arcs. When a robot arm stops, the appropriate arc is enabled and the arm's target state becomes 'rising'. This continues until the Maintain() fails and erases the entire automaton, reflecting the `do-watching` construct.

In general, an HCA corresponds to a tree of parallel processes whose execution follows the following routine. As this algorithm shows, a location is a simple process that asserts a target state and exits upon reaching that state or being aborted from above. Higher level HCAs manage child components and cannot be restarted until exiting.

```
Process Execute(HCA)
  If HCA is a location
    Assert target state until target reached
  Else
    For each initial child component M
      Start Execute(M)
    Repeat
      If the Maintain() condition fails then
        Abort each active child component
      For each child component M that exits
        For each transition $M \xrightarrow{C} N$ in HCA
          If C holds and N is not executing then
            Start Execute(N)
    Until no more child components are executing
  Exit
```

From a representational standpoint, team plans are similar to any other hierarchical plan. The only syntactic addition to turn a hierarchical plan into a team plan involves defining teams to perform activities and assigning roles to teammates. More precisely, injecting teamwork modeling into an existing hierarchical plan execution system involves adding three features (Tambe 1997):

- generalization of activities to represent team activities with role assignments;
- representation of team and/or sub-team states; and
- restrictions to only let a teammate modify a team state.

The key observation underlying the use of RMPL is how the language's approach to defining a control program as an HCA naturally matches the approach to defining a team plan with a model of flexible teamwork. Team plans are hierarchically defined in terms of sub-plans and primitive actions, where each teammate is assigned a role consisting of a subset of the sub-plans and actions. Returning to the our example, the Maintain() is a team HCA with components that are local HCAs for each rover. Thus the right hand rover need only address the components of Figure 7, and the two rovers need only communicate to be consistent over the team's Maintain() condition. The condition tells the rovers when to collectively abort their HCAs. In general, agents only need to communicate when a team level automaton changes its active components or some property of its Maintain() condition changes. Changes in a local HCA's components can be hidden.

As the example implies, all that we need to know to distribute an RMPL procedure is the distribution of mode variables and sensors. Since mode variable assignments were determined by DNNF distribution, the distribution of local and team HCAs follows from sensor and behavior distribution. Thus an RMPL programmer does not have to worry about synchronization issues across multiple agents. The underlying model of flexible teamwork will robustly manage these issues by keeping team state information consistent among the closely coordinated population of agents. The association of RMPL procedures to one or more robots similarly follows from the HCA distribution, and since RMPL procedures are used to implement plan activities, the distribution permeates to determining the local activities that a robot has which robots participate in which team activities.

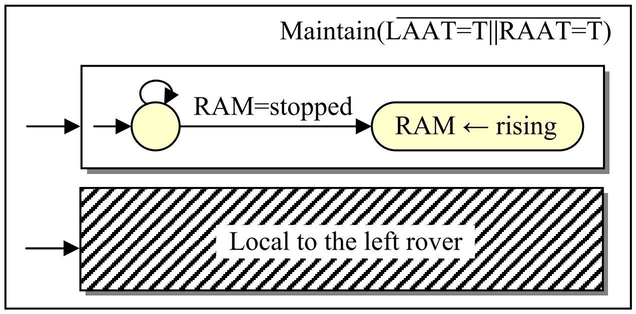

Figure 7. What the right-hand arm only executes those HCAs that refer to states and sensors local to the right arm.

Distributed Continual Planner

The need to rapidly respond to unexpected events motivates continuous planning, an approach where a planner continually updates a sequence in light of changing operating context. In such an operations mode, a planner would accept and respond to activity and state updates on a one to ten second time scale. CASPER (Chien *et al*. 2000) is an example of a continuous planner based on a heuristic iterative repair approach toward planning (Zweben *et al*. 1994, Fukunaga *et al*. 1997). This approach takes a complete plan at some level of abstraction and manipulates its actions to repair detected flaws. Example flaws would involve an action being too abstract to execute or many simultaneous actions with conflicting resource needs.

Extending the CASPER continual planning framework to perform continual iterative repair within the planner/scheduler modules results in the distributed CASPER continual planning algorithm illustrated in Figure 8. Essentially the algorithm manipulates the activities in PLAN to get rid of detected problems in PROECTION, and a window sweeps over the plan from left to right over to determine which near term activities to pass to the executives. While some activities are local, others are shared and must be kept identical across multiple planners.

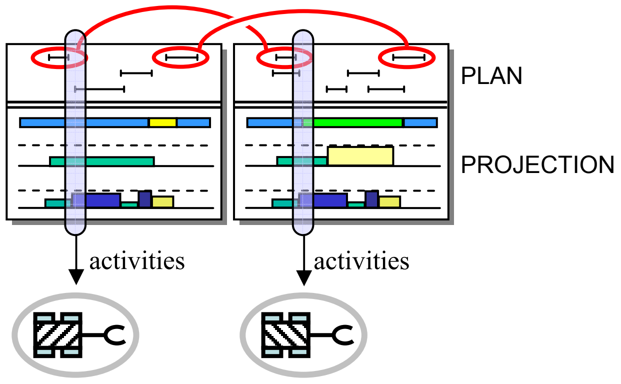

Figure 8. Two CASPER continuous planners that reason about an intersecting set of activities for two robots, where the intersection activities must be kept identical as the planners manipulate their plans.

More precisely, the algorithm is a continuous look of the seven steps below, where a mission manager can insert new goal activities into PLAN. Line 1 makes the PROJECTION variable (containing state variable profiles) always reflect how the spacecraft's state should evolve as its plan executes, and the sixth line causes this execution by passing near-term activities to the executive/diagnostician.

The expected state evolution changes as a plan gets new goal activities and the perceived state diverges from expectations. This divergence is caused by unexpected exogenous events and activities having unexpected outcomes. Since a planning model can only approximate the reality experienced during execution, these unexpected state changes can always happen. Thus revising PROJECTION can result in detecting flaws in a local plan at any moment, and lines 2 through 4 select and apply repair methods to fix these flaws. For instance, a satellite observation can take an unexpectedly long time to complete. Depending on the delay, a later observation may be impossible due to the target being too far behind the satellite when the observation starts. A repair method might fix the flaw by rescheduling the observation at a later time.

While changes to local activities can remain private to a single spacecraft, changes to shared activities must be made public, and line 5 communicates these changes. In general, the use of CONSTRAINTS and flaw repair methods can implement a number of coordination strategies. To be more concrete, suppose that a repair method suggests changing the start time of a shared activity. After changing the activity's *startTime* variable, the spacecraft has to inform other participating spacecraft of the change. In this way the spacecraft can follow an asynchronous weak commitment (AWC) (Yokoo& Hirayama 1998) search approach to maintaining plan coordination.

Distrubuted CASPER continual planning process

Given: a PLAN with multiple local and shared activities
a PROJECTION of PLAN into the future
a set of CONSTRAINTS on shared activities

1. Revise PROJECTION using the currently perceived state, new goal activities from a mission manager, and received changes to shared activities.
2. Heuristically choose a plan flaw in PROJECTION.
3. Heuristically choose a flaw repair method that honors CONSTRAINTS.
4. Use method to alter CONSTRAINTS, PLAN, and PROJECTION.
5. Communicate changes to shared activities in PLAN and CONSTRAINTS.
6. Release relevant near-term activities in PLAN to the executive.
7. Go to 1.

More precisely, the AWC approach, which is central to our architecture for managing shared activities, involves agents asynchronously assigning values to their variables from domains of possible values, and communicating the values to neighboring agents with shared constraints. Each variable has a non-negative integer priority that changes dynamically during search. A variable is consistent if its value does not violate any constraints with higher priority variables. A solution is a value assignment in which every variable is consistent.

To simplify describing the algorithm, suppose that each agent has exactly one variable and constraints between variables are binary. When an agent's variable value is not consistent with neighboring variable values, there can be two cases: (i) *good* case where there exists a consistent value in the variable's domain; (ii) *nogood* case that lacks a consistent value. In the *good* case with one or more value choices available, an agent selects a value that minimizes conflicts with lower priority agents – the *min-conflict* heuristic. On the other hand, in the *nogood* case, the priority of the agent is increased to *max+1,* where *max* is the highest priority of neighboring agents. In this approach, a bad value selection by a higher agent make does not force lower agents to exhaustively search for local solutions; nogood situations locally increase a priority to make previously higher agents choose new values. Furthermore, an increasing agent sends a *nogood message* with its *agentView* (the values and priorities of neighboring agents). These messages are used to avoid repeating past situations where an agent has no consistent values in its domain. Extension of AWC to multiple variables per agent has been investigated in (Yokoo and Hirayama 1998).

Related Work

Since distributed autonomy is an active field, a complete list of related systems would fill a tome, which forces me to be incomplete here. The most obviously related work is that by Simmons *et al.* (2002) in that it puts three layers on each robot and components at each layer communicate with each other, but their approach does not focus on characterizing the population of robots as a single entity and lacks a model-based approach toward mode estimation. On the other hand they include a capability server that enables adding robots to a population at any time, a feature that this approach currently lacks.

The closest related work on distributed sequencing of teams comes from STEAM (Tambe 1997) and TPOT-RL (Stone 1998). These two systems address teams of tightly coordinated agents that can fail, but they are based on rule-based approaches that lack system models to facilitate principled approaches to planning, mode estimation, and failure response.

Most other systems focus either on emergent behaviors of multiple closely coordinating behavioral agents or planned behaviors of loosely coordinated agents. Since this work involves planned behaviors of robots with varying levels of coordination, I will focus on such systems here.

Conclusions

This paper presents a model-based executive for command-ing teams of agents. It works by letting an operator define and command the team as a single entity with a single controlling CPU. A compiler then distributes the control functions guided by a specification assigning system components (sensors and actuators) to team members.

Each component of this system exists in isolation. CASPER is a very mature continual planner that has flown on the EO1 satellite, and SHaC (Clement&Barrett 2003) is an existing system for coordinating distributed planners based on shared activities that has been tested in simulation. The distributed executive (Barrett 2005) also has also been tested in simulation. Finally, the CAMPOUT distributed feedback controller (Pirjanian *et al.* 2001) has been tested on a variety of tasks from cooperative transport of a beam to cooperatively rappelling over a cliff. While each component of the system exists, the full integration of them has yet to occur.

Finally, humans may interact with the distributed autonomy system at each of the three levels: planner, executive, and feedback controller. At the feedback controller level, one of the sensors in figure 3 might be a human actuated joystick, giving the human tight control of low level behaviors. At the executive level a human might directly request the execution of a team procedure or perform assigned elements of procedures like any other agent. At the plan level, a human might manipulate shared activities that are subsequently altered in the planners on other agents. Thus from lowest to highest, each level has its mode of interaction, joy sticking to tactical to strategic.

Acknowledgements

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The author would also like to thank Seung Chung, Adnan Darwiche, Daniel Dvorak, and Mitch Ingham for discussions contributing to this effort

References

A. Barrett. "Model Compilation for Real-Time Planning and Diagnosis with Feedback." Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence. 2005.

S. Chien, R. Knight, A. Stechert, R. Sherwood, and G. Rabideau, "Using Iterative Repair to Improve Responsiveness of Planning and Scheduling," International Conference on Artificial Intelligence Planning and Scheduling, 2000.

B. Clement and A. Barrett. "Continual Coordination through Shared Activities." Proceedings of AAMAS-2003, Melbourne, Australia, July 2003.

A. Fukunaga, G. Rabideau, S. Chien, D. Yan 1997. "Towards an Application Framework for Automated Planning and Scheduling," In *Proceedings of the 1997 International Symposium on Artificial Intell-igence, Robotics and Automation for Space*, Tokyo Japan.

E. Gat. On three-layer architectures. In D. Kortenkamp, R. P. Bonnasso, and R. Murphy, editors, Artificial Intelligence and Mobile Robots. MIT/AAAI Press, 1997.

M. Ingham, R. Ragno, B. C. Williams. “A Reactive Model-based Programming Language for Robotic Space Explorers,” Proceedings of iSAIRAS-2001, St-Hubert, Canada, June 2001.

P. Nayak and B. C. Williams. “Fast Context Switching in Real-time Propositional Reasoning,” Proceedings of AAAI-97, Providence, RI, 1997.

P. Pirjanian, T. Huntsberger, A. Barrett. “Represent-ing and Executing Plan Sequences for Distributed Multi-Agent Systems.” Proceedings of IROS-2001, Maui, HI, October 2001.

R. Simmons, T. Smith, M. B. Dias, D. Goldberg, D. Hershberger, A. Stentz, R. Zlot, “A Layered Architecture for Coordination of Mobile Robots”, in Multi-Robot Systems: From Swarms to Intelligent Automata, A. Schultz and L. Parker (eds.), Kluwer, 2002.

P. Stone.. *Layered Learning in Multi-Agent Systems: A Winning Approach to Robotic Soccer*, MIT Press, Cambridge, MA 1998.

M. Tambe, “Towards Flexible Teamwork.” Journal of Artificial Intelligence Research, Volume 7. 1997.

B. C. Williams and P. Nayak. “A Model-based Approach to Reactive Self-Configuring Systems.” Proceedings of AAAI-96, Portland, OR, August 1996.

B. C. Williams, S. Chung, V. Gupta. “Mode Estimation of Model-based Programs: Monitoring Systems with Complex Behavior” Proceedings of IJCAI-2001, Seattle, WA, 2001.

M. Yokoo and K. Hirayama, “Distributed Constraint Satisfaction Algorithm for Complex Local Problems,” International Conference on Multi-Agent Systems, 1998.

M. Zweben, B. Daun, E. Davis, and M. Deale, “Scheduling and Rescheduling with Iterative Repair,” in *Intelligent Scheduling*, Morgan Kaufman, San Francisco, 1994.