A Model-Based Executive for Commanding Robot Teams

Anthony Barrett

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

Abstract. The paper presents a way to robustly command a system of systems 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 systems are both globally responsive by virtue of a team oriented interaction model and locally responsive by virtue of a distributed approach to model-based fault detection, isolation, and recovery.

1 Introduction

Over the next decades NASA mission concepts are expected to involve growing teams of tightly coordinated spacecraft in dynamic, partially understood environments. In order to maintain team coherence, each spacecraft must robustly respond to global coordination anomalies as well as local events. Currently techniques for implementing such teams are extremely difficult. They involve either having one spacecraft tightly control the team or giving each spacecraft separate commands with explicit communication actions to coordinate with others. While both approaches can handle two or three simple spacecraft, neither scales well to larger populations or more complex spacecraft. New techniques are needed to facilitate managing populations of three or more complex spacecraft.

Flexible teamwork 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 encoding their group activities. As Figure 1 shows, a user models a team as if one member

tightly controls the others, but a compiler takes that model and distributes it 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 having a user explicitly reason about the distribution.

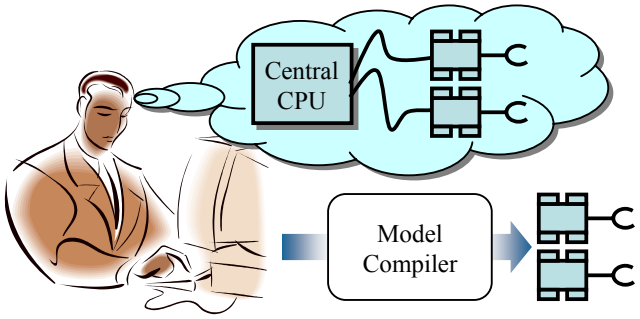

Fig 1. User models system assuming a central controller, and a compiler handles the underlying distribution and coordination.

Defining the approach starts with an explanation of the system architecture that resides on each component and how the components interact. Sections 3 and 4 then discuss the team sequencing and distributed mode management aspects of the system respectively. Section 5 subsequently discusses related work followed by conclusions in section 6.

2 System Architecture

The system architecture involves three distinct distributed components: mode identifier, mode reconfigurer, and team sequencer. As illustrated in Figure 2, the mode identifier combines sensory information from the hardware with past commands from the reconfigurer to determine the mode of the system. The mode reconfigurer in turn 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 system can take manually generated commands, it was initially motivated as an executive that supports distributed autonomy via shared activities [1].

While this architecture has been explored for single agent systems using TITAN [2], it has yet to be cleanly extended to tightly coordinated teams. This work makes the extension by developing distributed techniques for mode identification, mode reconfiguration, and team sequencing. In each component the techniques motivated replacing the underlying algorithms to provide the original capabilities while supporting distributed computation and hard real-time performance guarantees.

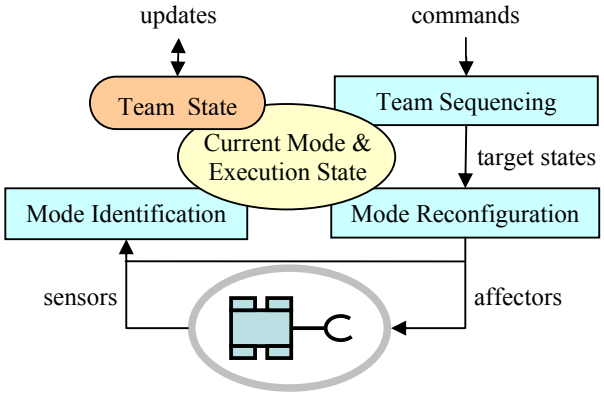

Fig 2. The component architecture of executive on each robotic element consists of three elements that current state information – some of which is identified as team state information and has to be kept consistent among all robotic elements.

3 Team Sequencing

The distributed onboard sequencer is based both on the Reactive Model-based Programming Language (RMPL) [3] and a model of flexible teamwork [4] developed within the distributed artificial intelligence community. This 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 a corresponding breakdown in teamwork. 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 the emergence of unexpected interactions caused either by slight timing delays or anomalies.

3.1 Procedural Control

RMPL elevates the level at which a control programmer thinks about a robotic systems. Instead of reasoning about sensors, actuators, and hardware, a RMPL programmer thinks in terms of commanding a system through a sequence of configuration states. As such, 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, consider the control program below for a simplistic approach toward getting to 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 conditional branching, iteration, concurrent task accomplishment, and preemption.

```
(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) [2], where the nesting of automata directly corresponds to the nesting of RMPL constructs. For instance, Figure 3 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.

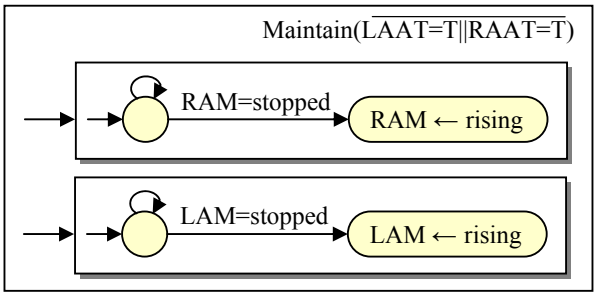

Fig 3. Hierarchical constraint automaton (HCA) for two rovers lifting a bar consist of four locations inside nested automata, where LAAT, RAAT, LAM, and RAM respectively denote `LArmAtTop`, `RArmAtTop`, `LArm.Mode`, and `RArm.Mode`.

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. Whenever 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 algorithm below. 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 their child components and cannot be restarted until exiting. This algorithm differs from the one presented in [2] due to maintaining a more hierarchical agent focus, which facilitates subsequent distribution. Cycling through all of the processes on each state change and recording the exits at the end of a cycle to subsequently enable transitions on the next cycle would make the two become identical. This divergence was made to facilitate distributed execution of an HCA.

```
Process Execute(HCA)
  If HCA is a location then
    Assert target state until target reached
  Else
    For each initial child component M
      Start Execute(M)
    Repeat
      Wait for a change to the local/team state
      If the Maintain() condition fails then
        Abort each active child component
      Else
        For each child component M that just exited
          For each transition M --C--> N in HCA
            If C holds and N is not executing then
              Start Execute(N)
    Until no more child components are executing
  Exit on end of cycle.
```

3.2 Teamwork Extensions

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 [4]:

- 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 4, 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 illustrates, all an RMPL programmer has to do to facilitate distributed sequencing is associate state variables with teammates to determine local and team HCAs. He 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.

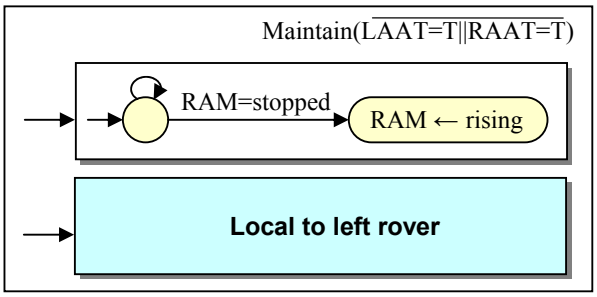

Fig 4. The right rover executes an HCA that ignores local information regarding the left rover.

4. Distributed Mode Management

By building off of the model-based device description language developed for DS-1's Mode Identification & Recovery (MIR) executive [5], 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 others' and ultimately sensor observations, which facilitates 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 component types are defined in terms of how a component's modes define constraints among its port variables, and these constraints are encode using *variable logic equations* – Boolean equations where the literals are simply variable equality constraints. For instance, the following defines an extremely simplistic system with two components for its robot arms using the language defined in [6].

```
(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 [7], where components are translated into Boolean equations. In both cases the systems 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 easily guarantee performance in all cases.

Instead of trying to prove the problem dependent speed of heuristics, we will take an approach suggested by knowledge compilation research. This approach involves moving as much of the computation into the 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 Decomposable Negation Normal Form (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 with two subtrees like Figures 6 and 7 for the two robot arms, and these subtrees are combined with an "and" node. In general, the compilation results look like Figure 5 with a substructure for each agent, and these substructures are combined in the team state.

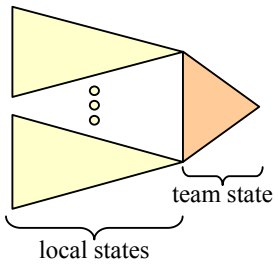

Fig 5. The global structure of compiled model consists of substructures local to each team member that are connected through a structure in a shared team state.

4.1 Mode Estimation

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 0, infinity, or some other value then respectively return default assignments, failure, or descend from the root to determine and return the variable assignments that contribute to its cost.

For instance, Figure 6 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 of zero is used to drill down to find the lowest cost tree with the mode assignment (i.e. RAM_1=stopped).

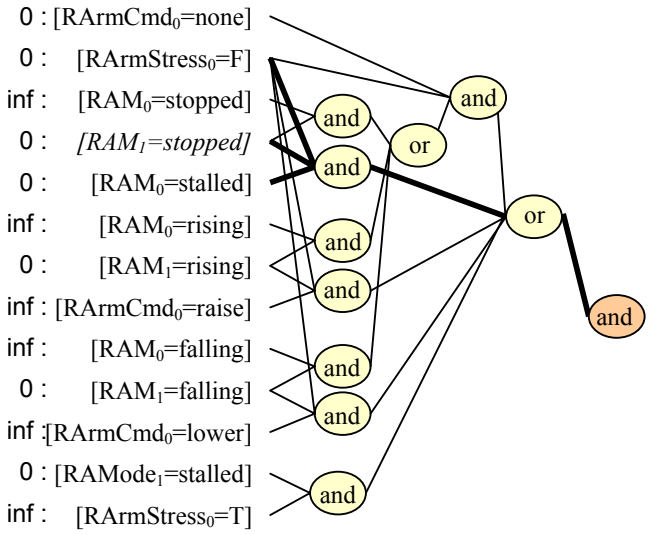

Fig 6. Utilizing local reasoning on its local and team model, the right-hand rover can determine that it has transitioned to a stopped mode.

Distributing the DNNF equations across a population is a simple matter of assigning sensors to agent and then assigning each node to an agent depending on the locations of sensors that contribute to that node's computation. If all contributers are on a single agent, then the node can be locally computed. Other nodes contribute to the team state. While all of the higher level nodes can be managed in the team state, it is more efficient to minimize the nodes in the team state. One way to reduce the number of nodes is to exclude nodes whose costs can be computed from other nodes in the team state. This results in only including the leftmost nodes of Figure 5's the team state structure. In any case, given the team state component of the structure, any agent can drill down to determine its local modes.

4.2 Mode Reconfiguration

It turns out that not only do components have modes to estimate, but they also accept commands to change modes to a target configuration once an estimate is determined. For instance, the simplistic robot arm model has four modes. In general, each component is modeled as a state machine that takes commands to transition between states and each state determines interactions among variables. In the example the robot arm can be commanded to rise or drop, but it stalls once arm stress is detected. This stalled mode subsequently puts the arm into a stopped mode once the stress is relieved.

While DS-1's MIR executive was able to perform real-time reconfiguration planning with this approach to representing models, it required a modeler to conform to four requirements: only consider reversible control actions, unless the only effect is to repair failures; each control variable has an idling assignment that appears in no transitions and each transition has a non-idling control condition; no set of control conditions for a transition is a proper subset of control conditions for another transition; and the components must be totally orderable such that the effects of one component has no impact on previous components. Alternative approaches based on universal planning [8] avoid these restrictions by taking a model and a target state and generating a structure that is used to generate commands to reach the target state in real-time regardless of the current state. Unfortunately universal plans are restricted to determine actions for reaching a single target state, but a robot will tend to have an evolving target state as it performs its commands. Also, universal plans tend to grow rapidly with system size.

This system both avoids the DS-1 restrictions and the universal plan limitations by taking a user supplied parameter *n* and guaranteeing to find an optimal plan from the current state to a target state if such can be reached within *n* steps. This guarantee is facilitated by evaluating a universal(*n*) plan against the current configuration and target configuration [9].

Definition 2: A universal(*n*) plan is a structure that can be evaluated in linear time to generate an optimal *n* level plan to reach any target configuration from any current configuration if such a plan exists.

Universal(*n*) plans are more general than universal plans by virtue of their not being tied down to a specific target configuration. They are more restricted than universal plans by virtue of the *n* level requirement, where a level is any number of simultaneous non-interacting actions. When increasing *n*, the universal(*n*) plan becomes less restrictive until reaching some model dependent value M – where there is a guarantee that target configuration can be reached from any configuration within M steps. In practice *n* is kept relatively small because universal(*n*) plans tend to grow rapidly with *n*.

To provide an example, Figure 7 uses a DNNF equation to represent a universal(1) plan for the two robot arm example. Just as in mode estimation, using the DNNF to determine the next command is a three-step process. First the current and target modes are used to assign costs to the RAM_0 and RAM_1 leaves respectively, and the $RArmCmd_0$ command leaves get user supplied command costs. Second, costs are

propagated to the tree root. Third, the root node's cost of 0 is used to drill down to find the actual command to perform. In this simplistic case the current right arm mode was stopped and, the target mode was rising, and the found command to pass to the right arm was raise.

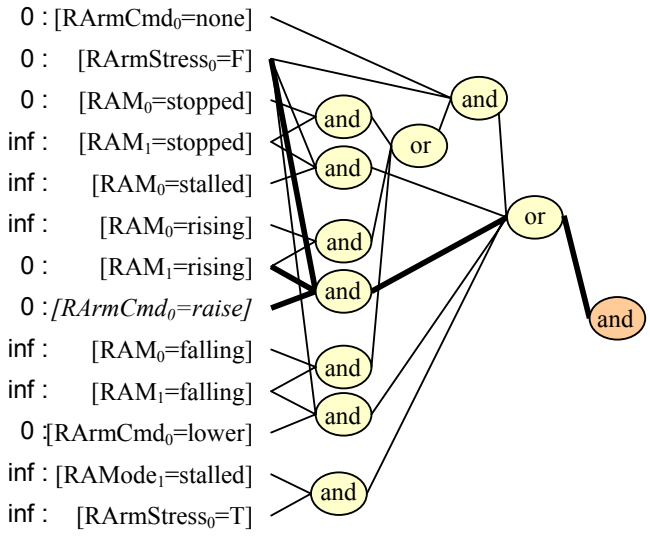

Fig 7. Utilizing local reasoning on its local and team model, the right-hand rover can determine how to make the arm rise.

5. Related Work

The closest related work on distributed sequencing comes from STEAM [4], MONAD [10], and TPOT-RL [11]. 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 mode estimation and failure response. Work by Stolzenburg and Arai [12] takes a more model-based approach by using Statecharts to specify a multiagent system, but they focus on communication via events as opposed to maintaining team state information. Still, the constructs of RMPL can be defined in terms of compilation to Statechart fragments instead of HCAs to facilitate formal analysis.

While others have made the leap to applying compilation techniques to both simplify and accelerate embedded computation to determine a system's current mode of operation, they are more restricted than this system. First, DNNF equation creation and evaluation was initially developed in a diagnosis application [13], but the resulting system restricted a component to only have one output and that there cannot be directed cycles between components. Our system makes neither of these restrictions. The Mimi-ME system [14] similarly avoided making these restrictions, but it can neither support distributed reasoning nor provide real-time guarantees by virtue of

having to collect all information in one place and then solve an NP-complete problem, called MIN-SAT, when converting observations into mode estimates. Our approach both supports distribution and real-time guarantees.

The closest related work on real-time reconfiguration planning comes from the Burton reconfiguration planner used on DS-1 [5] and other research on planning via symbolic model checking [15]. In the case of Burton our system improves on that work by relaxing a number of restricting assumptions. For instance, Burton required the absence of causal cycles, but our system has no problem with them. On the other hand, our system can only plan *n* steps ahead where Burton did not have that limitation. Similarly, the work using symbolic model checking lacked the *n*-step restriction, but it compiled out a universal plan for a particular target state. Our system uses the same compiled structure to determine how to reach any target state within *n* steps of the current state.

Finally, distribute behavioral systems like CAMPOUT [16] solve similar problems, but lack mechanisms for error handling. Such systems form a natural layer below the system presented here for teams with joint activities that are too tightly interacting to allow reasoning about mode management.

6. Conclusions

This paper presents a model-based executive for commanding 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.

As the example suggests, there are several ways to improve the system. From a representational perspective, the assignment of an agent to a role in a group activity is hardwired. For instance, there is no way to represent the possibility that `LArm` and `RArm` are interchangeable. The multi-agent community has explored multiple techniques for role assignment, but work needs to be done to include them in the team sequencer.

Also, knowledge compilation approaches like those used in mode management are not perfect. While onboard computation has linear complexity, that complexity is in terms of compiled DNNF equation size. Some problems are inherently intractable and lead to equations that are exponentially larger than the source model, but in practice that should never happen with engineered designs. Designs that result in inherently intractable mode estimation problems would be too uncontrollable to use in practice. As a rule of thumb, a system's mode estimation difficulty rises with the number of unobserved component interactions. Thus the number of interactions increases the size of the DNNF equation, but the number of sensors decreases it. Since engineers currently simplify estimation difficulty by adding sensors to a design, DNNF compilation results can be used to guide sensor placement if desired.

Finally, the system has only been tested in toy scenarios like this paper's running example. The main evaluation metrics are the size of the DNNF equation generated by the compiler and the size of the computed team state. Initial experiments in toy domains as well as a domain for a formation flying interferometer [17] show that the

size of the teamstate component of the DNNF equation depends on the complexity of the robot interactions and not on the complexity of the entire system. This bodes well for scaling issues.

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 Alan Oursland, Seung Chung, Adnan Darwiche, Milind Tambe, Daniel Dvorak, and Mitch Ingham for discussions contributing to this effort

References

1. Clement, B., Barrett, A.: "Continual Coordination through Shared Activities." In *Proceedings of the Second International Conference on Autonomous Agents and Multi-Agent Systems*, 2003.
2. Ingham, M., Ragno, R., and Williams, B. C.: "A Reactive Model-based Programming Language for Robotic Space Explorers." In *Proceeding of the International Symposium on Artificial Intelligence, Robotics and Automation in Space*, June 2001.
3. Williams, B. C., Chung, S., and Gupta, V.: "Mode Estimation of Model-based Programs: Monitoring Systems with Complex Behavior." In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*. August 2001.
4. Tambe, M., "Towards Flexible Teamwork." In Journal of Artificial Intelligence Research, Volume 7. 1997
5. Williams, B. C., Nayak, P. "A Model-based Approach to Reactive Self-Configuring Systems." In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*. August 1996.
6. Barrett, A. "Model Compilation for Real-Time Planning and Diagnosis with Feedback." In *Prodeedings of the Nineteenth Interantional Joint Conference on Artificial Intelligence*. July 2005
7. Nayak, P., Williams, B. C. "Fast Context Switching in Real-time Propositional Reasoning," In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, July 1997.
8. Schoppers, M. "The use of dynamics in an intelligent controller for a space faring rescue robot." *Artificial Intelligence* 73:175-230. 1995.
9. Barrett, A. "Domain Compilation for Embedded Real-Time Planning." In *Proceedings of the Fourteenth International Conference on Automated Planning & Scheduling*, June 2004.
10. Vu, T., Go, J., Kaminka, G., Veloso, M., Browning, B. "MONAD: A Flexible Architecture for Multi-Agent Control." In *Proceedings of the Second Interna-*

tional Joint Conference on Autonomous Agents and Multi-Agent Systems. July 2003.

11. Stone, P. *Layered Learning in Multi-Agent Systems: A Winning Approach to Robotic Soccer*, MIT Press, Cambridge, MA 1998.
12. Stolzenburg, F., Arai, T. "From the Specificatio of Multiagent Systems by Statecharts to Their Formal Analysis by Model Checking: Towards Safety-Critical Applications." In: Schillo, M. et al. (Eds.): MATES 2003, Lecture Notes in Computer Science, Vol. 2831. Springer-Verlag Berlin Heidelberg (2003). 131-143.
13. Darwiche, A. "Compiling Devices: A Structure-Based Approach," In *Proceedings of the Sixth International Conference on Knowledge Representation and Reasoning (KR)*. June 1998.
14. Chung, S., Van Eepoel, J., Williams, B. C. "Improving Model-based Mode Estimation through Offline Compilation," In *Proceedings of the International Symposium on Artificial Intelligence, Robotics and Automation in Space*, June 2001.
15. Cimatti, A., Roveri, M. "Conformant Planning via Model Checking." In: Biunido, S., Fox, M. (eds.): Recent Advances in AI Planning, 5th European Conference on Planning. Lecture Notes in Computer Science, Vol. 1809. Springer-Verlag, Berlin Heidelberg (2000). 21-34.
16. Pirjanian, P., Huntsberger, T., Barrett, A. "Representing and Executing Plan Sequences for Distributed Multi-Agent Systems." In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, November 2001.
17. Chung, S., Barrett, A. "Distributed Real-time Model-based Diagnosis." In *Proceedings of the 2003 IEEE Aerospace Conference*, March 2003.