ENHANCING SURVIVABILITY WITH DISTRIBUTED ADAPTIVE COORDINATION
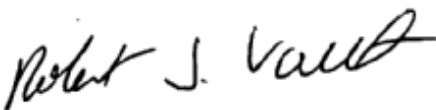
University of Massachusetts at Amherst

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-142 has been reviewed and is approved for publication.

APPROVED: *Robert J. Vaeth*

ROBERT J. VAETH
Project Engineer

FOR THE DIRECTOR: *signature*

WARREN H. DEBANY Jr., Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE JUNE 2003	3. REPORT TYPE AND DATES COVERED Final Jun 97 – Jun 00

4. TITLE AND SUBTITLE
ENHANCING SURVIVABILITY WITH DISTRIBUTED ADAPTIVE COORDINATION

6. AUTHOR(S)
Victor Lesser

5. FUNDING NUMBERS
C - F30602-97-1-0249
PE - 62702F
PR - F160
TA - 40
WU - 34

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Massachusetts at Amherst
Department of Computer Science
140 Governors Drive
Amherst Massachusetts 01003-9264

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency   AFRL/IFGB
3701 North Fairfax Drive   525 Brooks Road
Arlington Virginia 22203-1714   Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2003-142

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Robert J. Vaeth/IFGB/(315) 330-2182/ Robert.Vaeth@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 Words)*

The focus of this effort was to develop distributed detection and diagnosis algorithms for use in recognizing and explaining the cause of unacceptable performance of a distributed, multi-agent system. The explanation generated by the diagnosis algorithms was to be used by other components of the agent to reorganize processing in order to improve performance given current capabilities and resources. In this way, the system would have a higher degree of survivability in the event of software errors, hardware malfunctions, or hostile attacks. The researchers view survivable systems as computational organizations that can redesign themselves in response to threats and opportunities. A central assumption of organizational design is that there exist alternative ways to accomplish tasks in terms of agents, methods and resources used. In systems of any complexity, such alternatives do exist, and systems constructed for survivability will intentionally contain them. Under these conditions, the central challenges of survivability are making effective use of the available alternatives, acquiring knowledge about those alternatives, and making inferences based on that knowledge.

14. SUBJECT TERMS
Distributed Systems, Agents, Adaptive Coordination

15. NUMBER OF PAGES
52

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

Subject Terms:

Adaptive multi-agent systems; diagnosis; fault detection

Project Summary:

The researchers view survivable systems as computational organizations that can redesign themselves in response to threats and opportunities. A central assumption of organizational design is that there exist alternative ways to accomplish tasks in terms of the agents, methods, and resources used. In systems of any complexity, such alternatives do exist, and systems constructed for survivability will intentionally contain them. Under these conditions, the central challenges of survivability are making effective use of the available alternatives, acquiring knowledge about those alternatives, and making inferences based on that knowledge.

The are three technologies that need to be developed to address this view of survivable systems: (1) distributed coordination mechanisms that enable systems to organize themselves and accomplish critical tasks with available resources; (2) learning techniques that enable systems to improve with experience and model themselves and their environment, and thus creating the knowledge necessary for organizational design; and (3) detection and diagnosis algorithms that enable systems to use models that detect important changes and diagnose why those changes occurred. Detection and diagnosis provides the capability to effectively use learned knowledge for organizational design.

The focus of this work is the development of domain-independent detection and diagnosis algorithms that operate in an environment where agents are in some known distributed coordination pattern and where there is knowledge about the performance characteristics of agents. The agent's capabilities and behavior patterns are expressed in a domain-independent language called TÆMS. It is designed to model the problem-solving activities of an intelligent agent operating in environments with deadlines and limits on resource usage, where the information required for the optimal performance of a computational task may not be available, where the results of multiple agents' computations (to interdependent subproblems) may need to be aggregated in order to solve a high-level goal, and where an agent may be contributing concurrently to the solution of multiple goals. The TÆMS approach is based on explicitly modeling what is known and uncertain about agent goals, hierarchical task structures, alternative actions, sets of resources, and external agent capabilities.

As part of this effort, in order to provide an appropriate framework for evaluation, a Multi-Agent Survivability Simulator (MASS) has been developed to provide a concrete, deterministic, and welldefined environment suitable for testing multi-agent coordination, negotiation, and diagnosis algorithms. MASS was created to test various coordination mechanisms by allowing the elements of the system to detect, react and adapt in the face of adverse working conditions. Each agent has its own local view of the world and its own goals, but is capable of coordinating these goals with respect to remote agents. To accurately model these complex systems, an environment is needed which permits the simulation of an agent's method execution. To this end, a distributed event-based simulator has been developed that is capable of simulating the effects that directed attacks and/or a capricious environment have on agent method execution and recovery. Features of the current implementation include support for an arbitrary number of mixed agents, scriptable event and task generation, and controllable resource limitations and behavior.

Included with this report are three papers that represent the major output of the research project. The first paper, entitled "Diagnosis as an Integral Part of Multi-Agent Adaptability," describes how agents working under real world conditions may face an environment capable of changing rapidly from one moment to the next, either through perceived faults, unexpected interactions or adversarial intrusions. To gracefully and efficiently handle such situations, the members of a multi-agent system must be able to adapt, either by evolving internal structures and behavior or repairing or isolating those external influences believed to be malfunctioning. The first step in achieving adaptability is diagnosis — being able to accurately detect and determine the cause of a fault based on its symptoms. This paper examines how domain-independent diagnosis plays a role in multi-agent systems, including the information required to support and produce diagnoses. Particular attention is paid to coordination-based diagnosis directed by a causal model. Several examples are described in the context of an Intelligent Home environment, and the issue of diagnostic sensitivity versus efficiency is addressed.

The second paper, entitled "Using Self-Diagnosis to Adapt Organizational Structures," illustrates another use of diagnosis for adaptability. The specific organization used by a multi-agent system is crucial for its effectiveness and efficiency. In dynamic environments, or when the objectives of the system shift, the organization must be able to change as well. Results from experiments employing such a system in the Producer-Consumer-Transporter domain are also presented.

The last paper, entitled "An Agent Infrastructure to Build and Evaluate Multi-Agent Systems: The Java Agent Framework and Multi-Agent System Simulator," discusses the simulation system used for experiments on multi-agent adaptability. It describes their component-based Java Agent Framework (JAF) used for rapidly building different types of agents, and addresses the issues encountered in designing a suitable environmental space for evaluating the adaptive qualities of multi-agent systems.

Chronological List of Publications Resulting from this Research Project

1. Horling, Bryan; Benyo, Brett; and Lesser, Victor. Using Self-Diagnosis to Adapt Organizational Structures. In *Proceedings of the 5th International Conference on Autonomous Agents*, ACM Press, pp. 529-536, Montreal, June 2001. Also published as University of Massachusetts/Amherst CMPSCI Technical Report 1999-64, November 1999.
2. Raja, A.; Wagner, T.; Lesser, V. "Reasoning about Uncertainty in Agent Control." In *Proceedings of the 5th International Conference on Information Systems, Analysis, and Synthesis, Computer Science and Engineering: Part 1*, Volume VII, pp. 156-161, Orlando, FL, 2001.
3. Raja, Anita; Wagner, Thomas; and Lesser, Victor. "Reasoning about Uncertainty in Design-to- Criteria Scheduling." In *Proceedings of AAAI 2000 Spring Symposium on Real-Time Autonomous Systems*, AAAI Press, pp. 76-83, Stanford, CA, March 2000.
4. Raja, Anita; Lesser, Victor; and Wagner, Thomas. "Toward Robust Agent Control in Open Environments." In *Proceedings of 4th International Conference of Autonomous Agents* (AA 2000), pp. 84-91, Barcelona, Spain, June 2000. Also published as University of Massachusetts/Amherst CMPSCI Technical Report 1999-059.
5. Vincent, Regis; Horling, Bryan; Lesser, Victor. "Experiences in Simulating Multi-Agent Systems Using TÆMS." In *The Fourth International Conference on MultiAgent Systems (ICMAS 2000),* Boston, MA: AAAI Press, pp. 455-456. Also published as University of Massachusetts/Amherst CMPSCI Technical Report 1999-75.

6. Wagner, T.; Lesser, V. "Relating Quantified Motivations for Organizationally Situated Agents." In *Intelligent Agents VI: Agent Theories, Architectures, and Languages,* N.R Jennings & Y. Lesperance (eds.). Berlin: Springer-Verlag, 2000. Vol. 1757, pp. 334–349. Also published as University of Massachusetts/Amherst CMPSCI Technical Report 1999-21.

7. Xuan, P; Lesser, V; Zilberstein, S. "Communication in Multi-Agent Markov Decision Processes." Extended abstract in *Proceedings of the Fourth International Conference on Multi- Agent Systems* (ICMAS'2000), Boston, MA: AAAI Press, pp. 467-468. Also published as University of Massachusetts/ Amherst CMPSCI Technical Report 2000-01.

8. Xuan, Ping; Lesser, Victor. "Incorporating Uncertainty in Agent Commitments." In *Intelligent Agents VI: Agent Theories, Architectures, and Languages,* N.R Jennings & Y. Lesperance (eds.). Berlin: Springer-Verlag, 2000. Vol. 1757, pages 57-70.

9. Xuan, Ping; Lesser, Victor; Zilberstein, Shlomo. "Formal Modeling of Communication Decisions in Cooperative Multi-agent Systems." In *Proceedings of the Second Workshop on Game Theoretic and Decision Theoretic Agents*, 2000.

10. Horling, Bryan; Lesser, Victor; Vincent, Regis; Bazzan, Ana; and Xuan, Ping. "Diagnosis as an Integral Part of Multi-Agent Adaptability." In *Proceedings of DARPA Information Survivability Conference and Exposition*, IEEE Computer Society. pp. 211-219, Hilton Head, SC, January 2000. Also published as University of Massachusetts/Amherst CMPSCI Technical Report 1999- 03.

11. Lesser, V.; Atighetchi, M.; Benyo, B.; Horling, B.; Raja, A.; Vincent, R.; Wagner, T.; Xuan, P.; and Zhang, S.X.Q. "A Multi-Agent System for Intelligent Environment Control." In *Proceedings of the Third International Conference on Autonomous Agents*, New York, NY: ACM Press, 1999, pp. 291-298.

12. Horling, Bryan; and Lesser, Victor. "Using Diagnosis to Learn Contextual Coordination Rules." In *Proceedings of the AAAI-99 Workshop on Reasoning in Context for AI Applications,* AAAI Press, pp. 70-74, Orlando, FL, July 1999. Also published as University of Massachusetts/Amherst CMPSCI Technical Report 99-15.

13. Jensen, David; Atighetchi, Michael; Vincent, Régis; Lesser, Victor. "Learning Quantitative Knowledge for Multiagent Coordination." In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI-99)*, American Association for Artificial Intelligence, pp. 24-31, Orlando, FL, August 1999. Also published as University of Massachusetts/Amherst CMPSCI Technical Report 1999-04.

14. Lesser, Victor; Atighetchi, Michael; Benyo, Brett; Horling, Bryan; Raja, Anita; Vincent, Regis; Wagner, Thomas; Xuan, Ping; Zhang, Shelley XQ. "The Intelligent Home Testbed." In *Proceedings of the Autonomy Control Software Workshop* (Autonomous Agent Workshop), Seattle, 1999.

15. Wagner, Thomas; Shapiro, Jonathan; Xuan, Ping; Lesser, Victor. "Multi-Level Conflict in Multi- Agent Systems." In P*roceedings of the AAAI-99 Workshop on Negotiation in MAS*, AAAI Press, pp. 50-55, Orlando, FL, April 1999. Also available as University of Massachusetts/Amherst CMPSCI Technical Report 99-17.

16. Xuan, Ping; Lesser, Victor. "Handling Uncertainty in Multi-Agent Commitments." University of Massachusetts/Amherst CMPSCI Technical Report 1999-05, 1999.

17. Horling, Bryan. "A Reusable Component Architecture for Agent Construction." In University of Massachusetts/Amherst CMPSCI Technical Report 1998-49, October 1998.

18. Raja, Anita; Lesser, Victor; and Wagner, Thomas. "A More Complex View of Schedule Uncertainty Based on Contingency Analysis." In University of Massachusetts/Amherst CMPSCI Technical Report 98-04, February 1998.

19. Sugawara, T. and Lesser, V. "Learning to Improve Coordinated Actions in Cooperative Distributed Problem-Solving Environments." In *Machine Learning,* 33, 1998, pp. 129–153.

20. Vincent, Regis; Horling, Bryan; Wagner, Tom; Lesser, Victor. "Survivability Simulator for Multi-Agent Adaptive Coordination." In *International Conference on Web-Based Modeling and Simulation*, Volume 30, Number 1. Fishwick, P., Hill, D. and Smith R. (eds.), The Society for Computer Simulation International, pp. 114-119, San Diego, CA, 1998.

21. Wagner, Thomas; Raja, Anita; Lesser, Victor. "Modeling Uncertainty and Its Implications to Design-to-Criteria Scheduling." University of Massachusetts/Amherst CMPSCI Technical Report 1998-51.

22. Lander, S.; and Lesser, V. "Sharing Meta-Information to Guide Cooperative Search Among Heterogeneous Reusable Agents." In *IEEE Transactions on Knowledge and Data Engineering*, Volume 9, Number 2, 193-208, 1997.

Diagnosis as an Integral Part of Multi-Agent Adaptability *

Bryan Horling, Victor Lesser, Régis Vincent, Ana Bazzan, Ping Xuan
Department of Computer Science
University of Massachusetts

Abstract

Agents working under real world conditions may face an environment capable of changing rapidly from one moment to the next, either through perceived faults, unexpected interactions or adversarial intrusions. To gracefully and efficiently handle such situations, the members of a multi-agent system must be able to adapt, either by evolving internal structures and behavior or repairing or isolating those external influenced believed to be malfunctioning. The first step in achieving adaptability is diagnosis - being able to accurately detect and determine the cause of a fault based on its symptoms. In this paper we examine how domain independent diagnosis plays a role in multi-agent systems, including the information required to support and produce diagnoses. Particular attention is paid to coordination based diagnosis directed by a causal model. Several examples are described in the context of an Intelligent Home environment, and the issue of diagnostic sensitivity versus efficiency is addressed.

Content Areas: Intelligent Agents : Tasks or Problems : Multi-Agent Communication or Coordination or Collaboration; Intelligent Agents : Tasks or Problems : Learning and Adaptation

Overview and Motivation

Designing systems utilizing a multi-agent paradigm offers several advantages. They can easily utilize distributed resources, work towards multiple goals in parallel, and reduce the risk of a single point of failure.

One common failing of multi-agent systems, however, is brittleness. In the face of an adverse or changing environment, assumptions which designers commonly make during the construction of these complex networks of interacting processes may become incorrect. For instance, the designer may assume a certain transfer rate over the local network, or that a particular method may produce results of a given quality, both of which can easily change over time. In today's networked environment, it is also quite possible for adversarial intrusions to occur, capable of producing an even wider range of symptoms. Failure to act on incorrect assumptions can lead to degraded performance, incorrect results, or in the worst case even bring the entire system to a halt.

One solution is for the designer to be extremely paranoid, and essentially over-engineer each aspect of the system to reduce the chance of failure (e.g. always over-coordinate, execute redundant plans). Although such precautions certainly have their place in computing systems, they can also increase overhead to undesirable levels. This overhead then decreases the overall level of efficiency, as the system essentially wastes effort avoiding failures which would not have occurred in any case.

To improve the robustness of multi-agent systems, without unduly sacrificing efficiency, the system should be designed with adaptability in mind. An agent working under changeable conditions must have knowledge of its expected surroundings, goals and behavior - and the capacity to generate new, situation-specific coordination plans when these expectations are not met. The key to initiating adaptive behavior, therefore, is detecting such failed expectations, and determining their cause so they may be corrected. We believe that this need can be satisfied, and thus adaptability promoted, by incorporating domain and coordination independent diagnosis capabilities into the individual agents within the multi-agent system.

In this paper we will discuss how diagnosis plays a role in the adaptability of an intelligent home multi-agent system (Lesser *et al.* 1999). In this environment, major appliances, such as the dishwasher, waterheater, air conditioner, etc., are each controlled by an individual autonomous agent. The intelligent home provides

us with interesting working conditions, allowing the creation of scenarios involving constrained resources, conflicting objectives and cooperative interactions. We believe the issues raised by the intelligent home model can be sufficiently generalized to apply to multi-agent systems operating in other domains.

To overview the role diagnosis may play in a multi-agent system, consider the following scenario. A dishwasher and waterheater exist in the house, related by the fact that the dishwasher uses the hot water the waterheater produces. Under normal circumstances, the dishwasher assumes sufficient water will be available for it to operate, since the waterheater will attempt to maintain a consistent level in the tank at all times. Because of this assumption and the desire to reduce unnecessary network activity, coordination is normally unnecessary between the two agents. Consider next what happens when this assumption fails, perhaps when the owner decides to take a shower at a conflicting time (i.e. there is an assumption that showers only take place in the morning), or if the waterheater is put into "conservation mode" and thus only produces hot water when requested to do so. The dishwasher will no longer have sufficient resources to perform its task. Lacking diagnosis or monitoring, the dishwasher could repeatedly progress as normal but do a poor job of dish-washing, or do no washing at all because of insufficient amounts of hot water. Using diagnosis, however, the dishwasher could, as a result of poor performance observed through internal sensors or user feedback, first determine that a required resource is missing, and then that the resource was not being coordinated over. By itself, this would be sufficient to produce a preliminary diagnosis the dishwasher could act upon simply by invoking a resource coordination protocol. After reviewing its assumptions, later experiences or interactions with the waterheater, it could refine and validate this diagnosis and perhaps update its assumptions to note that hot water should now be coordinated over, or that there are certain times during the day when coordination is recommended. It should be clear that even simple diagnostics capabilities can provide a fair measure of robustness under some circumstances.

This work represents the continuation of our previous work on diagnosis for single agent (Hudlická & Lesser 1987) and multi-agent systems (Hudlická *et al.* 1986; Bazzan, Lesser, & Xuan 1998; Sugawara & Lesser 1993; 1998). The emphasis of this new work is to show that diagnosis can be exploited for a wider set of issues than indicated in our earlier work and more importantly that this diagnosis can be done in a domain-independent manner. The use of diagnosis is also a recent theme in the work by Kaminka and Tambe (Kaminka & Tambe 1998). Our work differs in the set of issues we are interested in diagnosing. For example, in Kaminka, collaborative agents use one another reflectively based on plan recognition, as sources of comparative information to detect aberrant behavior due to, for example, inconsistent sensing by different agents of environmental con-

ditions. Our work in contrast is mainly concerned with the performance issues surrounding the use of situation-specific coordination strategies. In this view of coordination, the strategy used for agent coordination must be tailored to specifics of the current/expected environment and the coordination situations that an agent will encounter. Over or under coordination can be harmful, respectively, because of the expenditure of resources to implement a coordination strategy that doesn't contribute sufficiently to a more efficient use of resources or the lack of appropriate coordination that leads to suboptimal use of resources. Implicit in this discussion is that there are a set of assumptions about agents behaviors and availability of resources that is the basis for effective situation specific coordination. Detection in this case involves recognizing when a monitored performance based assumption is no longer valid; diagnosis is then taking this triggering symptom and determining the detail set of assumptions about agent and resource behavior that is responsible for the symptom in the current context.

Several facets of agent diagnosis will be covered in this paper by detailing a diagnosis system that we have implemented in the Intelligent Home environment. What sort of information is needed? How can the diagnosis be produced? How does the diagnosis affect adaptability? These questions will be addressed using the diagnosis framework we have created, which focuses on the resources, coordination and activities associated with the agent. In the following section, the use of assumptions and organizational knowledge, along with fault detection techniques, will be discussed. Our diagnosis-generating framework, incorporating and utilizing this information, will then be introduced. Several examples detailing diagnosis in the Intelligent Home will provide further motivation and functional details about our system. We will then provide a more quantitative analysis of detection and diagnosis sensitivity tradeoffs, and conclude with directions for future research.

Information Requirements

For diagnosis to function, some sort of knowledge about expected behavior must be known a priori, to serve as a basis for comparison. The exact information which is needed depends on the diagnosis capabilities needed by the agent, but the data typically fits into one of three categories: knowledge about the agent's expected operational behavior, including environmental assumptions; methods for detecting deviations from those expectations; and faculties for diagnosing these deviations when they are found.

Expectations and Assumptions

For diagnosis to function, some information must be available describing what the correct, or at least expected, behavior of the agent should be. We have found that a great deal of useful method execution and goal
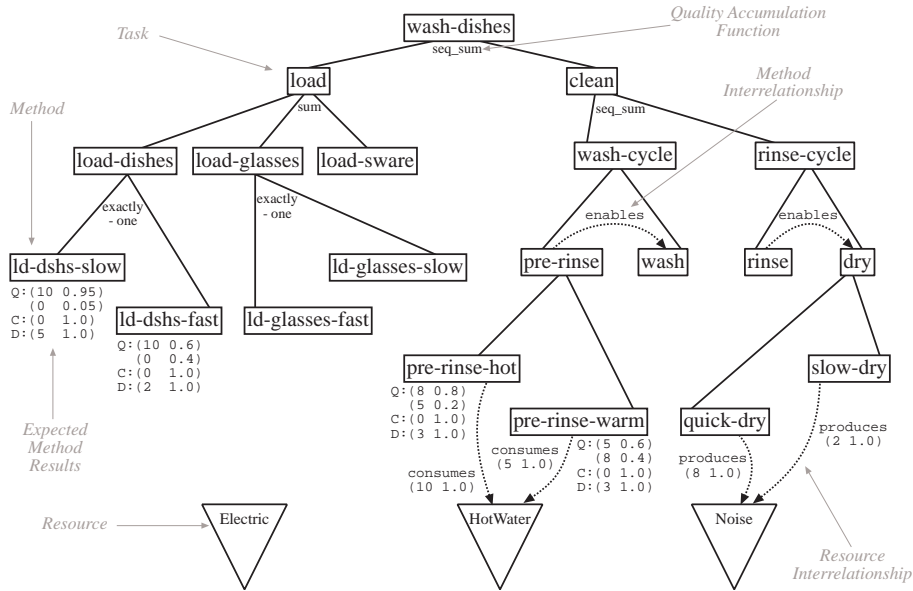
Figure 1: Abbreviated TÆMS task structure example for the dishwasher agent

achievement information can be succinctly encoded in a domain-independent way with a goal/task decomposition language called TÆMS (Decker & Lesser 1993). A graphical example of the dishwasher's TÆMS task structure can be seen in Figure 1. TÆMS provides an explicit representation for goals, and the available subgoal pathways able to achieve them; each branch in the tree terminates at an executable method. Expected method behavior and interactions between other methods and resources may also be represented explicitly. Associated with each method is a distribution-based description of its expected quality, cost and duration measures. These descriptions are represented by the Expected Method Results in the figure, where each possible method result along each trait consists of expected value-probability pair. Similarly, the effects of hard (A enables or disables B), soft (A facilitates or hinders B) and resource (A produces or consumes resource B) interrelationships are also quantitatively described with distributions.

In our discussion so far we have not focused on the underlying coordination architecture. Our initial thinking was that diagnosis needed to be strongly tied to the specifics of this architecture, so initial efforts (Bazzan, Lesser, & Xuan 1998) were thus bound to the GPGP coordination architecture (Decker & Lesser 1995). However, our stance has changed on this matter. We now feel that so long as TÆMS is a faithful representation of the expected local agent behavior and its interaction with other agents and resources, and that information is provided about what aspects of TÆMS were used in the current coordination context, and what was the actual schedule and results of activities executed, diagnosis can be done largely in a domain and coordination independent structure.

Another set of information, describing pertinent external assumptions, is needed for the diagnosis system to reason about the agent's interactions with its environment. Such characteristics currently used in our agents include network behavior (e.g. bandwidth, latency), entities thought to exist external to the agent (e.g. entities one might interact with), and resource characteristics (e.g. low/high bound, usage patterns), each of which has a direct correlation with how the agent should coordinate with other agents in the system. Organizational knowledge, the information an agent has about where and how it fits into its society, is a subset of this category. It may be useful, for instance, for the agent to have a record of the types of agents it is expected to interact with, and what sort of interactions should take place. In the the introductory example, this sort information let the dishwasher know that there was a prior assumption that coordination over hot water was unnecessary.

Detecting Possible Failures

Once descriptive information and models are incorporated into the agent, the process of using that information to detect possible inconsistencies becomes relevant. Consider the simple case of detecting abnormal method results. Within the TÆMS structure, the expected cost, quality and duration outcomes are described for each method. Armed with this information, the diagnosis system can use a light-weight comparison monitor to determine when a deviation from expected behavior might have occurred. The dishwasher used the method-resource relationship description in its TÆMS task structure, for instance, to determine that hot water was necessary for its dish-washing task to successfully complete. Performance threshold assumptions can
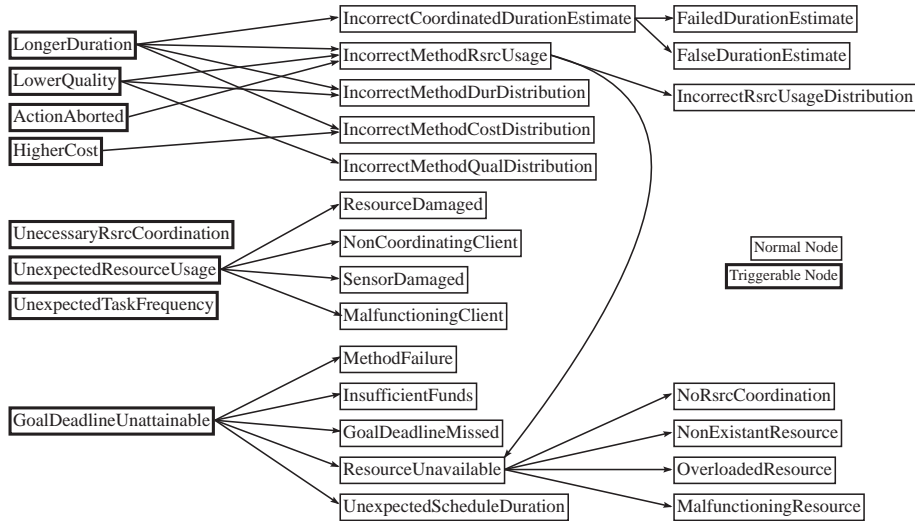
Figure 2: Example causal model structure for diagnosis in the Intelligent Home

then be used to determine the severity of the deviation, which would help determine the correct diagnosis and response later on. On-line learning of models can also be used to track long term trends in behavior, which can help determine if a deviation is caused by a fundamental shift in the environment or is just a one-time aberration.

Using knowledge about method interactions, also found in the TÆMS model, the diagnosis system can determine if a failure might have been brought about by some other method which had or had not successfully executed. If the method's source is local, an activity log can be checked for more evidence. If the offending method's source is remote, the list of known external agents can then be used to track down the culprit. A mixture of model based comparisons, combined with directed evidence gathering, provides a good base for detecting possible failures.

Performing the Diagnosis

The agent can now use its expectation information and failure detection methods to begin the actual process of diagnosis. Diagnosis is a well-researched field, with many different methods and techniques already available to the system designer (W. Hamscher 1992). Our goal was to use a technique that offered great flexibility in the information it could use and the diagnoses it could generate, without sacrificing subject scope or domain independence. Recent work in the field of diagnosis (Kaminka & Tambe 1998; Toyama & Hager 1997) has shown that viable new technologies are still being developed. It is not clear, however, that any single diagnostic technique is suitable for the entire range of faults exhibited by multi-agent systems. It was therefore desirable to use a system or framework capable of incorporating different diagnostic techniques, i.e. make use of different specific methods for the types of failures they best address.

Expanding on work first researched in (Sugawara & Lesser 1993), we chose to organize our diagnostic process using a causal model. The causal model is a directed, acyclic graph organizing a set of diagnosis nodes. Figure 2 shows such a graph, constructed to address issues brought up by examples in this paper, which we have implemented for the Intelligent Home agents; complete graphs addressing broader topics can be found in (Bazzan, Lesser, & Xuan 1998). Each node in the graph corresponds with a particular diagnosis, with varying levels of precision and complexity. As a node produces a diagnosis, the causal model can be used to determine what other, typically more detailed, diagnoses can be used to further categorize the problem. Within the diagnosis system, the causal model then acts as a sort of road map, allowing diagnosis to progress from easily detectable symptoms to more precise diagnostic hypotheses as needed.

The causal model in Figure 2 focuses on coordination, behavior and resource issues. Within the diagram, several nodes have bold-faced outlines. These nodes are *triggerable*, meaning they periodically perform simple comparison checks to determine if a fault may have occurred. This trigger-checking activity is a primary mechanism for initiating the diagnostic process. The runtime usage of the causal model is shown when considering the diagnostic activity of the agent in the introductory example. In the example, the dishwasher has performed an activity, but achieved a lower than expected amount of quality in the results. Using the given causal model, the LowerQuality node would be triggered. The resulting diagnosis would activate the child nodes of LowerQuality: IncorrectMethodRsrcUsage, IncorrectMethodDurDistribution, and IncorrectMethodQualDistribution. The first node attempts to encapsulate the idea that something went wrong with the resources expected to be used by the method, while the latter two address possible discrepancies in

8

the method's expected duration and quality. Incorrect-MethodRsrcUsage would then produce a diagnosis, activating IncorrectRsrcUsageDistribution and ResourceUnavailable. The resource was not used, so ResourceUnavailable would be triggered, activating its four child nodes. Of these, NoRsrcCoordination (possibly in combination with OverloadedResource) would then determine the exact problem.

The causal model addresses each of the design goals previously mentioned. The automatic flow of diagnoses through the graph structure allows the designer to add or remove subgraphs and nodes at will to increase or decrease the diagnostic specificity offered by the model. Thus, although the model shown in this paper is targeted towards a specific set of faults, the causal model in general allows one to create a diagnostic system for any range of faults or intrusions who's set of symptoms that can be observed or deduced are differentiable from one another. In our implementation, each node in the model corresponds to an individual persistent diagnostic object, capable of passively listening for data or actively gathering evidence. This means that, within the causal framework, individual nodes are free to use whichever diagnostic technique is needed, offering a great deal of flexibility to the system designer. The persistent nature of the diagnosis object also allows for direct control over the amount of evidence required for a diagnosis to be triggered and produced, since diagnostic analysis can continue through several episodes. We will come back to this notion of diagnosis sensitivity and confidence in a later section.

Faults involving multiple symptoms (or lack thereof) can be handled elegantly by the causal model, by incorporating a single node for the fault which uses diagnostic evidence from several other symptom-verifying nodes. For instance, a node diagnosing incorrect local method behavior descriptions could be derived with a diagnosis from another node detecting methods which take too long to execute, in conjunction with a lack of diagnoses from nodes diagnosing competing theories. A detected fault may also be caused by the cumulative effects of several other deviations, which did not merit diagnosis individually. This can occur, for instance, when a goal deadline is missed because each in a series of method invocations took slightly longer than expected. Individually, the methods' durations were within their respective distribution ranges; an expected deadline produced with the mean of each of these distributions could very well be missed under these circumstances.

The designer is also free to make nodes as domain independent as possible, so with a carefully thought out structure it is possible to transport the model from one system to the next with little modification. In addition to being domain independent, we also hypothesize that such a structure may be coordination independent; accurate diagnoses can be formed about coordination activities independent of the protocol's details. The fact that the framework scales so easily in scope means that a common, domain independent core may also be used among a group of agents, which augment the model with more specific nodes to meet their individual diagnostic needs.

Example Diagnosis Generation

In this section we will go over several multi-agent scenarios from the Intelligent Home domain, and how diagnosis based adaptation plays a vital role in its successful behavior. The first example gives a complete agent-by-agent description of a malfunctioning agent scenario, while the remaining examples will sketch out the usefulness of diagnosis under other circumstances.
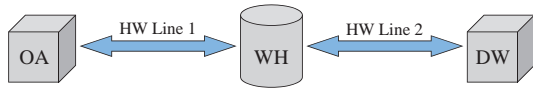
Detecting Software Failure

This scenario contains three agents, each with its own set of capabilities, goals and knowledge (see Figure 3), and a set of water pipes which connect them. The central figure in the scene is the waterheater, which produces hot water at a rate dictated by requests posed by the agents it serves. The owner of the house, being quite thrifty, has set the waterheater's goal to produce the minimum amount of hot water possible, thereby minimizing cost. The waterheater will therefore keep no minimum amount of water in the tank, forcing any agent needing hot water to coordinate over its usage (i.e. to explicitly schedule the production of hot water at a specific time). Elsewhere in the house is some other hot water-using appliance, which has a generic goal it needs to complete by a certain time. It is functioning normally, coordinating with the waterheater as needed.

The third agent, a dishwasher, has as its goal to wash the load of dishes currently in its possession. It has successfully started this operation, but through some sort of malfunction has arrived in a state where it is endlessly executing the method "pre-rinse-hot", a method which uses hot water (see Figure 1). The coordination component in the dishwasher still functions however, and therefore is also repeatedly coordinating over the hot water being used by the cyclic execution. As it happens, the dishwasher has also been set with a higher priority level than the other generic appliance working towards its goal. This has the end result of overwhelming the waterheater, forcing it to reject the hot water requests of the other appliance.

Lacking diagnosis, this scenario would end in failure. The dishwasher would never complete its load, the waterheater will produce much more water than would normally be needed, and the other appliance will fail to meet its goal deadline. Each agent, however, is equipped with information and a diagnosis model similar to that shown in Figure 2. Figure 3 should give the reader some notion of the characteristics each agent has which are relevant to this situation.

The dishwasher is first to detect a problem. The UnexpectedTaskFrequency node in its causal model is triggered, which proceeds to gather evidence on the current situation. Using the local activity log, schedule,

Figure 3: Software failure scenario overview

and assumptions about expected task frequency, it produces a diagnosis with a high confidence value. For this example, we will assume the dishwasher has no repair mechanisms for this problem, so no further actions are taken.

Eventually, the diagnosis component at the waterheater is also triggered. A root node UnexpectedResourceUsage is activated, which uses knowledge about expected hot water usage to determine a failure may be occurring. This node then activates each of its children in the causal model: ResourceDamaged, NonCoordinatingClient, SensorDamaged and MalfunctioningClient. Analysis of coordination logs, and sensor data from both the tank and output pipes rule out the first three possibilities. MalfunctioningClient proceeds to contact those agents using water line two - only the dishwasher in this case. The dishwasher sends the waterheater its diagnosis of UnexpectedTaskFrequency, which acts as supportive evidence for the MalfunctioningClient diagnosis. The waterheater, acting on this diagnosis, can either reduce the priority of the dishwasher's coordination requests, or cut off the flow of water into water line two.

The other appliance, unaware of the problems being handled by the waterheater, only knows that its coordination attempts have been consistently refused for quite a while. Realizing that its ability to meet its deadline requirement is in question, the diagnosis node GoalDeadlineUnattainable activates its child nodes: MethodFailure, InsufficientFunds, GoalDeadlineMissed, UnexpectedScheduleDuration and ResourceUnavailable. The ResourceUnavailable node can use the attempted schedule, local TÆMS task structure and coordination logs to determine that the hot water resource is unavailable. This in turn activates the child nodes NoRsrcCoordination, NonExistantResource, OverloadedResource and MalfunctioningResource. OverloadedResource gains evidence by querying for and receiving the UnexpectedResourceUsage and MalfunctioningClient diagnoses from the waterheater. At this point, the other appliance can reschedule with the knowledge that the hot water resource is overloaded. Further analysis by the OverloadedResource node could also detect when the problem had been resolved, allowing the other appliance

to continue using hot water as it becomes available.

Slight modifications to this example demonstrate how diagnosis can be used to detect aggressive intrusions into the multi-agent system. In the altered scenario, the dishwasher's logic has been compromised in such a way that it continually uses hot water without coordination. The activity exhibited by the remaining two agents could remain much the same, with the exception that the dishwasher would produce a NonCoordinatingClient diagnosis in lieu of MalfunctioningClient (since it is unlikely that the compromised dishwasher would admit to malfunctioning). The waterheater could unilaterally react to this diagnosis by cutting off the water supply to line two, a reasonable short term repair for this problem.

Over-coordination

One interesting efficiency scenario is that of over-coordination. A spectrum of coordination models is possible in multi-agent systems, ranging from fully explicit, verbose communication to "well-known" assumptions or implicit agreements. Clearly, it is more efficient to reduce inter-agent communication if possible, but how can an agent know when it is safe to do so? One method, similar to a technique described in (Toyama & Hager 1997), makes use of a persistent diagnostic process to monitor tested changes.

In this example, the UnecessaryRsrcCoordination node begins its work by monitoring the coordination which takes places over the system's resources. If it detects that requests for a particular resource are always being satisfied, it forms the hypothesis that it is not necessary to coordinate over that resource, either because it is automatically replenished (e.g. a low-bound maintaining waterheater) or a common resource lacking contention (e.g. electricity under normal circumstances). The problem solving component in the agent could then react to this diagnosis by ceasing coordination over that resource. Over time, the persistent diagnosis object then monitors this resource, to see if methods requiring it are affected, and adjusting the diagnosis as needed. This diagnosis can then provide the feedback necessary for the agent to maintain the situation-specific assumptions needed for it to be efficient in its environment. A more complicated diagnostic process could also further classify coordination relations, based on coordination type, temporal cycles or sensitivity to requested resource amounts.

Method Outcome Discrepancies

A key measure of adaptability is how the agent responds to unexpected results. If a method fails, does the system blindly reschedule, or does it take into account the reasons for the failure? How does the agent react when method performance varies within what is considered normal ranges?

The introductory example demonstrates a problem of this type. In the example, a NoRsrcCoordination diagnosis was used to target the perceived fault,

which allowed the agent to repair its original schedule rather than generate a new one which may have made the same mistake. A more interesting scenario involves an agent's reaction to different levels of discrepancies - when should an agent adapt to, ignore or repair a problem? The TÆMS task language allows the designer to explicitly encode the expected behavioral ranges a method may exhibit, and learning algorithms can be employed to maintain the structure as time passes. Results falling within these ranges are expected, and should not trigger diagnosis. The remaining issue requires more information to discriminate enough to make an intelligent choice. More detailed organizational knowledge about method behavior can be used to determine thresholds, allowing the agent to discriminate between acceptable and unacceptable variations in long term performance deviations. Diagnosis can then use this information and other available evidence to provide the specific problem description seen in other examples, which the problem solver can use to pick the appropriate course of action.

Consider an example involving method duration. An agent executes method X, expecting it take between 10 and 15 clicks to complete (a click being an arbitrary unit of time), as encoded in its task structure. In addition, the designer has specified in the organizational knowledge that durations up to 40 clicks are within "acceptable" tolerances, which is designed to take into account such things as network activity fluctuations or noisy sensor data. If X were to complete in 25 to 35 clicks, the agent would take note of the event and modify performance characteristics in TÆMS after determining the deviation was not caused by a fault other than inaccurate expectations (such as missing resources, hindering method interrelationships). Instead, X takes 100 clicks to complete, clearly outside of its expected range. This value is also well outside of the acceptable tolerances, so the agent should not adapt its expectations to this new situation. The aberration would then initiate diagnosis activity, which would monitor future behavior of this method. We will assume that the operating conditions match no other diagnosis' symptoms. Over time, as X is executed again, a clearer picture of its current performance could be generated, which can help determine if the failure was a single event, or the first instance of a software or hardware fault, or an intrusion.

Detection and Diagnosis Sensitivity

While diagnosing problems in a multi-agent setting is an interesting problem in its own right, it is also important to examine the effect of detection and diagnostic frequency on overall system behaviors. Specifically, one may wonder what the appropriate level of "aggressiveness" is for detection and diagnosis. On one hand, if the process is very sensitive, effort may be wasted monitoring behaviors operating normally, or adapting to faults that don't exist. On the other hand, a more skeptical diagnostic system may ignore triggers signifying larger problems, or spend so much time gathering evidence
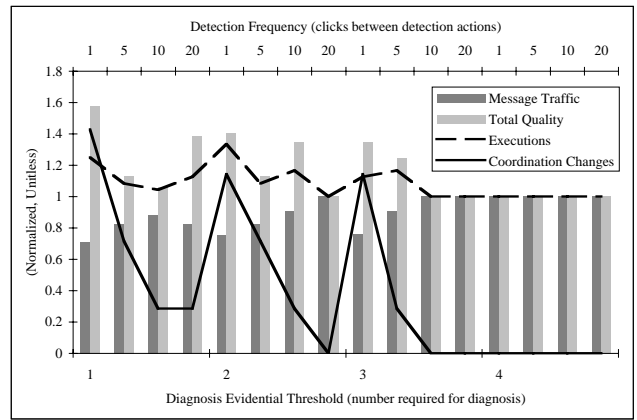

Figure 4: Results from the over-coordination scenario

and improving confidence that the eventual adaptation comes too late.

The notion of inappropriate coordination will be used to more closely examine this continuum. In the experiment, a water-using appliance and the waterheater interact. The water-using appliance executes a schedule using different amounts of water, at different times. The waterheater attempts to maintain a pre-specified level of water in the tank, and so will react to unanticipated usage. The heater is also able to coordinate over water usage, thus ensuring that the water is available when it is needed. The challenge here is to determine when the water should be coordinated over: should the agent expend energy to ensure the needed water is present, or should it simply use the water and accept the fact that there is some probability of failure due to lack of resources?

To diagnose the situation, the water-using appliance examines the coordination pattern with the waterheater, and the resulting effects when coordination is stopped. Thus, if the agent determines it is almost always receiving positive responses from its coordination requests, it may opt to stop coordinating with the hypothesis that the waterheater can reactively keep up with its requests. If, however, this is not the case, negative execution results will prompt the agent to begin coordinating again.

Two parameters are varied in the different experiments: the frequency at which the coordination and result data is examined, and the amount of evidence required for a diagnosis to be produced (e.g. for a change in coordination to take place). The scenario is allowed to run for a set amount of time, after which the number of coordination messages, overall quality, number of method executions, and number of coordination changes are recorded. An optimal agent would thus minimize the number of coordination attempts and maximize quality. The number of coordination changes and executions is not good or bad in and of itself, but is indicative of the rate and results of adaptation the agent has exhibited.

The results of the experiment can be seen in Figure 4. The lower X axis measures the amount of evidence required for the agent to create the diagnosis which would effect coordination decisions. The upper X axis should be viewed as four separate cases, each of which measure the effects of decreased detection frequency. The data shown is otherwise unitless and normalized, to help accentuate the observed trends.

The overall trend of the graph indicates that as more pieces of diagnostic evidence are required, or if the detection frequency is decreased, the agent will be more conservative in its coordination decision. To see this trend, look at the rate of coordination change first along the lower axis, representing increases in required evidence, then in each of the four subsections indicated by the upper axis, which shows decreases in detection frequency. In each case, the rate of coordination change decreases.

The effects of this coordination decision are shown by the bars in the graph. At higher rates of change (e.g. the left side of the graph and sub-graphs), the agent tends to communicate less, while at the same time producing more quality. This may at first seem contradictory - one would expect that an agent which coordinated more would have a higher quality, since it is supposedly increasing the probability that the required water will be available. What is not considered, however, is the time cost of coordination. When coordinating, the agent both wastes time waiting for coordination responses, thereby delaying the start of execution, and may also decide not to execute at all, when its coordination request is denied. These two items combine to reduce the rate of execution under coordination (shown by the Executions line), which in turn reduces the total amount of quality. Thus, this data suggests that under these environmental and behavioral circumstances, the diagnosis component should be restricted to a narrow window of data, which is continuously analyzed, allowing the agent to quickly react to changes.

The results shown here are meant to persuade the reader that an important continuum exists in the sensitivity of diagnostic components, rather than claim that any one point is reasonable for all, or even this particular type of fault. Clearly the optimal point in this range is dictated both by the techniques being employed and the circumstances under which they are used. Designers of diagnosis-capable agents should keep this trade-off in mind, so as to avoid counteracting the benefits of adaptability with unnecessary overhead.

Current Implementation & Future Work

The diagnosis architecture described in this paper has been implemented, and used in several coordination and behavior fault based scenarios. We have over a dozen agents operating in the Intelligent Home, with varying levels of diagnostics ability and ad hoc adaptability. The causal model shown in this paper has also been implemented, and extensions to other software, hardware and intrusion based faults are being considered.

In the future, we plan on more closely addressing the eventual reaction to diagnosis: adaptation and repair mechanisms. Specifically, the implementation, domain independence and quantitative analysis of these mechanisms will be considered. We also plan to more thoroughly analyze the efficiency cost of adapting to non-fault conditions.

Conclusion

Adaptation can be an important part of any computational system, but the susceptible of multi-agent systems to broad classes of computational, behavioral and adversarial faults make it especially vital. We believe that a robust and flexible diagnostic component, coupled with informative models and data, is a necessary part of this adaptive capability. Agents capable of self and remote diagnosis will play an important role in making multi-agent systems both robust and efficient.

References

Bazzan, A. L.; Lesser, V.; and Xuan, P. 1998. Adapting an Organization Design through Domain-Independent Diagnosis. Computer Science Technical Report TR-98-014, University of Massachusetts at Amherst.

Decker, K. S., and Lesser, V. R. 1993. Quantitative modeling of complex environments. *International Journal of Intelligent Systems in Accounting, Finance, and Management* 2(4):215–234. Special issue on "Mathematical and Computational Models of Organizations: Models and Characteristics of Agent Behavior".

Decker, K. S., and Lesser, V. R. 1995. Designing a family of coordination algorithms. In *Proceedings of the First International Conference on Multi-Agent Systems*, 73–80. San Francisco: AAAI Press.

Hudlická, E., and Lesser, V. R. 1987. Modeling and diagnosing problem-solving system behavior. *IEEE Transactions on Systems, Man, and Cybernetics* 17(3):407–419.

Hudlická, E.; Lesser, V., P.; J.; and Rewari, A. 1986. Design of a distributed diagnosis system. UMASS Department of Computer Science Technical Report 86-63.

Kaminka, G. A., and Tambe, M. 1998. What is wrong with us? improving robustness through social diagnosis. In *in Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*. AAAI.

Lesser, V.; Atighetchi, M.; Horling, B.; Benyo, B.; Raja, A.; Vincent, R.; Wagner, T.; Xuan, P.; and Zhang, S. X. 1999. A Multi-Agent System for Intelligent Environment Control. In *Proceedings of the Third International Conference on Autonomous Agents*. Seattle,WA, USA: ACM Press.

Sugawara, T., and Lesser, V. 1993. Learning control rules for coordination. In *Multi-Agent and Cooperative Computation '93*, 121–136.

Sugawara, T., and Lesser, V. R. 1998. Learning to improve coordinated actions in cooperative distributed problem-solving environments. *Machine Learning*. To appear.

Toyama, K., and Hager, G. D. 1997. If at first you don't succeed... In *in Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*. AAAI.

W. Hamscher, L. Console, J. d. K., ed. 1992. *Readings in Model-Based Diagnosis*. Morgan Kaufmann.

Appendix: Diagnosis Implementation Details

To allow the reader to better understand how diagnosis currently functions in the agent, the implementation details of several of the causal model nodes will be discussed in this appendix.

LongerDuration

As can be see from Figure 2, the LongerDuration node is triggerable, meaning it periodically checks simple observable traits which might be symptoms of a larger problem. The node uses two pieces of information to perform this check: the list of currently executing methods and the local TÆMS task structure. During the trigger-check phase, the node will compare the current running time of each executing method to the expected time indicated by the task structure. If the current time exceeds the mean of the expected time plus some given threshold, the node becomes *activated*.

Once activated, a new instance of node is created which more closely monitors that single method whose duration seems to be too long. A different, presumably looser, standard is used to determine excessive duration at this point. This allows the triggering to be somewhat sensitive, while the actual diagnosis can allow for a wider range of performance. If this new threshold is also passed, a diagnosis will be generated noting the problem. If the method does not pass the new threshold, the instance will silently deactivate itself.

IncorrectMethodQualDistribution

As this node is not triggerable, it depends on other nodes for activation. Typically, this node will be activated by the LowerQuality node, which itself would be triggered by a method whose resultant quality is lower than expected (determined by a method similar to that described in the previous section).

IncorrectMethodQualDistribution, once activated, acts primarily as an interface to a local long-term learning component. The learning component will independently monitor all method invocations, and track their execution characteristics. As results are obtained, the learning component is able to build its own local version of the task structure, which can be used to verify or contradict the expected values present in the agent's actual task structure. If a sufficient amount of evidence has been gathered by the learning component, a chi-squared test is made to determine the significance of the differences (if any) between what it has learned and what was expected. If the difference is significant, a diagnosis is produced, otherwise the node silently deactivates itself.

The two nodes IncorrectMethodDurDistribution and IncorrectMethodCostDistribution work similarly.

NoRsrcCoordination

This node, typically activated when a method results differ from expectations, is used to determine if the symptoms could have been caused by a missing resource which was uncoordinated over. The diagnosis activating the node will contain a reference to the method which has misbehaved. By examining the agent's local task structure, the node can determine if the method in fact used resources by searching for the method-to-resource interrelationships which conventionally indicated such usage.

Agents in our environment actually posses two copies of its task structure, which are differentiated as subjective and conditioned views. The subjective view represents what the agent believes to be the true working conditions imposed by the environment. Interrelationships included in this model indicate that relationships between methods and resource do exist, and will have impact on one another (so far as the agent knows). The conditioned view, however, represents only those relationships which the agent deems necessary to coordinate over. Thus, a method-to-resource relationship in the subjective view indicates the the agent is aware a certain method will interact with the resource; the presence of the same relationship in the conditioned view indicates the the agent has actively decided to coordinate over that interaction. Thus, the conditioned view is used as a coordination-independent view of the coordination activities the agent will exhibit at runtime.

By examining the dichotomy between the subjective and conditioned view, the NoRsrcCoordination node can determine both when resource interactions are taking place, and whether or not coordination took place over that interaction. When a poorly performing method is delivered to the node, it first determines if an interaction took place by looking for relationships in the subjective view. If such relationships don't exist, the node simply deactivates because there was no coordination that could have taken place. If the relationship does exist, however, and the corresponding relationship is not present in the conditioned view, the NoRsrcCoordination may pose a diagnosis indicating that poor performance may be a result of needed resources which were not coordinated over (and therefore may have not been available).

UnecessaryRsrcCoordination

This node is also triggerable, searching for possible instances where coordination may be unnecessary. The triggering activity in this case is whenever coordination

over a resource is performed for the first time, which is determined by listening to the event stream emanating from the coordination component. Once a coordination event is observed, the node proceeds to set up long term monitoring facilities to track the activity and results of this coordination.

Once in place, the monitors count the number of times a resource is attempted to be coordinated over, and the number of acceptances and rejections arising from these attempts. Once a certain amount of evidence (coordination attempts) have been made, the node examines the gathered data to calculate the probability of a coordinate attempt being accepted. If this ratio is above a certain threshold, a diagnosis is created indicating that coordination may be unnecessary.

If no coordination changes are made by the agent, the node will continue gathering evidence and updating its diagnosis when changes are observed. A sliding window of evidential data is maintained which helps prevent the node's diagnosis from being unduly affected by historical events. If, however, a coordination change is made, the node will response by throwing out all of its evidence, and begin listening to the NoRsrcCoordination node. Because no coordination is taking place, the monitors previously put in place will offer no new evidence as time passes. Instead, The results of the action will be the new, albeit indirect, source of evidence. By monitoring the NoRsrcCoordination node, UnecessaryRsrcCoordination can determine if its diagnoses adversely affects the activity of the agent, and can revise its diagnosis as necessary, which should allow the agent to know when and if coordination should be restored.

FailedDurationEstimate

This node is responsible for diagnosing when an agent performing a previously-coordinated over task fails to meet local and remote expectations (as opposed to FalseDurationEstimate, which attempts to determine when a remote agent misrepresents the duration). A good example of this is when an acting agent fails to achieve the duration estimate it submitted in a contract accepted through a contract-net protocol because of a local execution error (a required resource might not have been available, or the local distribution description might be incorrect).

The parent node, IncorrectCoordinatedDurationEstimate, has presumably already determined that the method in question was both coordinated over and exceeded the agreed upon duration. If this is the case, and the observed duration was greater than the agreed upon duration (if any), diagnosis takes place. FailedDurationEstimate node uses a form of distributed diagnosis to gather evidence by querying the executor for pertinent diagnoses it may have generated. If the executing agent reports a diagnosis concerning the method in question and its duration, the requesting node can infer that an execution error occurred, which resulted in the method taking longer to complete. A diagnosis

could be produced based on this information.

Using Self-Diagnosis to Adapt Organizational Structures *

Bryan Horling, Brett Benyo, and Victor Lesser
University of Massachusetts
Department of Computer Science
Amherst, MA 01003
{bhorling, bbenyo, lesser}@cs.umass.edu

ABSTRACT

The specific organization used by a multi-agent system is crucial for its effectiveness and efficiency. In dynamic environments, or when the objectives of the system shift, the organization must therefore be able to change as well. In this paper we propose using a general diagnosis engine to drive this process of adaptation, using the TÆMS modeling language as the primary representation of organizational information. Results from experiments employing such a system in the Producer-Consumer-Transporter domain are also presented.

Keywords: Organization and social structure, organization self-design.

1. OVERVIEW

As the sizes of multi-agent systems grow in the number of their participants, the *organization* of those agents will be increasingly important. In such an environment, an organization is used to limit the range of control decisions agents must make, which is a necessary component of scalable systems. Are agent agents arranged in clusters, a hierarchy, a graph, or some other type of organization? Are the agents' activities or behaviors driven solely by local concerns, or do external peers or managers have direct influence as well? Is communication between agents active, via messaging of some sort, or passive, using observations or engineered

assumptions? These and other characteristics define the *organizational structure* of a multi-agent system - the rules which define the roles agents play and the manners in which they interact with other agents in the system.

Clearly the characteristics described above will have an impact on the efficiency and responsiveness of both large and small multi-agent systems. It should also be intuitively clear that the effectiveness of the organization is dependent on the agents, environment, and goals involved in the system. The problem then, is how to derive such a structure given a particular situation. The simplest option is to statically define the organization when the system is developed. This has the benefit of being a simple and direct solution, but can become impractical when the sets of agents and goals are large and diverse. Static solutions also suffer when elements of the multi-agent system are dynamic, since characteristics of the environment, organizational goals, or member agents may change such that the initial organization becomes inefficient. Members of the agent pool may become deactivated or compromised in some way, making it impossible for the system to function correctly, or other agents may not be used effectively when they are added. In this sense, the organization is a set of assumptions that the system works by. As these assumptions become invalid, the organization must be able to adapt to keep the system viable.

The term Organizational Self-Design (OSD) has been used previously [2] to describe the general technique of employing the members of a multi-agent system to generate or adapt their own organizational structures at runtime. Earlier work in this area tended to focus on adapting specific qualities of the organization, such as task allocation [9] or load balancing [6, 8]. Organizational structure generation has also been proposed as arising from local [6], global [2], and hybrid [11] perspectives. Each of these systems demonstrated specific techniques that worked well and efficiently in their respective environments, but they were not general solutions to the problem. In this paper we propose a more general approach, using diagnosis, to detect deficiencies in the organizational model and assist in the creation of solutions to those deficiencies; the eventual goal being to create a reusable organizational adaptation engine. We will show how a general diagnosis engine, coupled with a powerful representation of that organization, can be used to effect change in a wide range of characteristics from arbitrary perspectives.

To help make this notion of organizational adaptation more concrete, we will look at an example from the Producer, Consumer, Transporter (PCT) domain [4]. In this domain, there are conceptually three types of agents: producers, which generate resources; consumers, which use them; and transporters, which move resources from one place to another. In general, a producer and consumer may actually be different faces of a factory, which consumes some
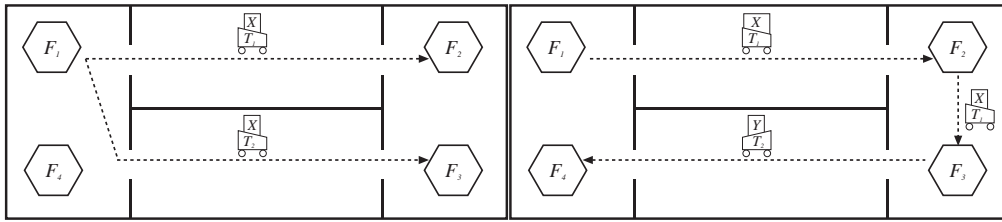
Figure 1: The initial (L) and revised (R) transporter organizations.

quantity of resources in order to produce others. There are several characteristics of this domain where alternatives exist for the factories and transporters - the choices made at these points by or for the PCT agents make up the organizational structure of the system. Examples of such characteristics include the types and quantities of resources a producer should generate, the set of potential sources a consumer should obtain required resources from, and what paths a transporter may choose to follow as it moves about.

In this example, consumers F_2 and F_3, shown in Figure 1, require some amount of resource X. In the initial organization, each is being supplied with X by producer F_1. X is then supplied to F_2 and F_3 by transporters T_1 and T_2, respectively, each of which operates at 50% capacity. Factory F_4 is initially idle, but at some future point in time it obtains production request, which requires resource Y to be satisfied. Fortunately, F_3 produces Y, but in the initial organization, no additional transporters are available to deliver the needed goods. With a diagnostic system in place, the transporters could determine that their initial organization, while functional under the initial conditions, included under-loaded transporters and was therefore potentially suboptimal. Instead of using two transporters running at 50% capacity, just one at 100% capacity could satisfy the original requirements for X expressed by F_2 and F_3, albeit at a slight time penalty because of the extra stop. Thus, if instantiated, the revised organization would leave T_2 free to perform the transportation required by F_4. More quantitative results from this domain will also be covered in section 4. Related work, using diagnosis to learn coordination rules in an intelligent home scenario, can also be seen in [5].

Figure 2 shows at a high level how we propose organizational design can be situated and integrated in an agent. In this architecture, critical components within the agent, such as those responsible for problem solving, negotiation and scheduling, obtain the vast majority of their information from an organizational design layer. This layer abstracts and filters elements of the operating environment in a manner consistent with the agent's role in the organization. The abstraction is composed of one or more information sources, such as TÆMS structures or MQ values [12], capable of encoding the various aspects of the organization. TÆMS a task and interaction modeling language, will be discussed in detail in a later section. MQ (motivational quantity) values, which give the agent a more powerful way to reason about the utilities of its tasks, will not be covered in this paper. To permit adaptation, the organizational design layer is maintained by a diagnostic subsystem, which attempts to repair faults and inefficiencies by adjusting elements of the organizational structure. This diagnostics process can itself be driven by a number of sources, including observations of the environment, conditions monitored within the agent, and discourse with other agents. The direct effects of these diagnoses typically take place within a relatively small group of agents, so one can think of this technique as being a search process for the correct organization through local adaptation. The organization will go through a set of

distribution adaptations, each involving a series of local adaptations by individual agents.

Going back to the previous PCT example, we can see how this technique would work in practice. The initial organizational structure would be encoded in TÆMS structures in both the transporters and factories. They would indicate such characteristics as what goals the factories and transporters should work towards and how they could be accomplished. Initially, the organization would be unconstrained, permitting the type of interactions seen in Figure 1L. Diagnosis running on the transporters or factories would determine that while the transporter loads were well balanced in the initial state, the arrangement was not necessarily the most efficient use of their abilities. F_2 could use this information to add a constraint to it's local organizational representation, indicating that it should use T_1 for its transportation needs. Later, when F_4 requests the use of a transporter, T_2 will then be available.

In the next section we will give more details about our view of the actual knowledge used by an agent to represent the organizational information that makes up the abstraction layer shown in Figure 2. Following this, we will cover our diagnostic system, shown in the middle of this same figure, and how it is integrated into and used by our agents. In section 4 results from an experiment in the PCT domain will be covered, and in section 5 we will present our conclusions.

2. ORGANIZATIONAL KNOWLEDGE

As mentioned in the previous section, the range of information that comprises an organizational structure can be quite broad. It is our opinion that there is no single, comprehensive set of characteristics that might make up *the* definition of an organizational structure. Instead, the set is dependent more on what alternatives are possible within a particular multi-agent system and which of those alternatives can have an impact on the system's behavior and effectiveness. Given that, we will present in this section our organizational representation, called TÆMS (Task Analysis, Environmental Modeling, and Simulation), which is flexible enough to model a wide range of organizational characteristics.

2.1 TÆMS

The primary representation of the organizational structure is done with the domain-independent TÆMS task modeling language [3] (see Figure 3 for a simple example). A TÆMS task structure is essentially a goal decomposition tree, where leaf nodes represent executable primitive methods and internal task nodes provide a hierarchical organization. Root level tasks (those with no supertasks) are known as task groups, and conceptually represent high level goals that might be achieved. Associated with each task is a quality-accumulation function (QAF), which indicates how the quality of the task is calculated from that of its subtasks. Associated with each method is a distribution-based description of its expected quality, cost and duration measures. Together, the probabilistic method de-
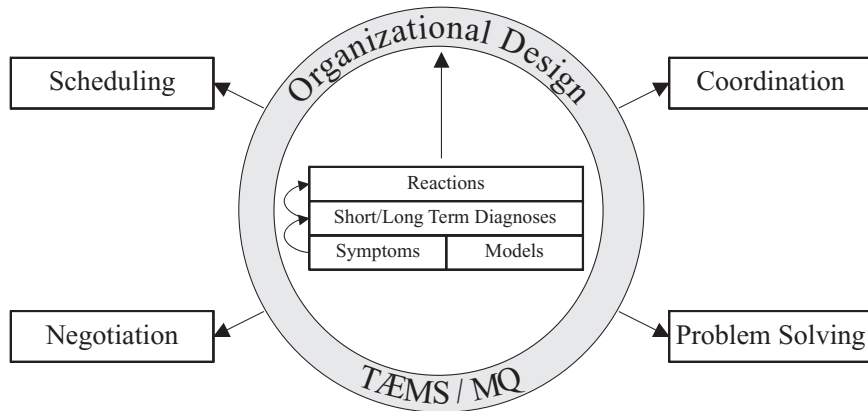
Figure 2: Role of organizational knowledge within an agent.

scriptions and QAFs allow a scheduler to effectively reason about the traits and tradeoffs of a wide range of possible schedules. A third type of element, interrelationships, which arise between internal tasks, methods and resources can be used to indicate a wide range of interactions, such as enables, facilitates, hinders and produces (e.g. performing a task will *enable* the execution of another, or a task will *produce* some amount of resource, as seen in Figure 3). Interrelationships may also span task structures between agents, and tasks and methods performed by remote agents may be represented locally. Combined, the capabilities give developers using TÆMS the flexibility to model a wide range of traits, from low-level performance characteristics of a single action to a high-level representation of the system's control hierarchy.

TÆMS task structures are typically used to encode the different mechanisms for achieving a goal, and the constraints and tradeoffs associated with each potential plan. They are also used to describe both the potential capabilities of an agent and the subset of those capabilities it should employ given its place in the organization. To do this, each agent will have two different versions, called *views*, of its local task structure: *subjective* and *conditioned*. The subjective view contains what the agent believes to be the complete model of its local execution alternatives[1]. The conditioned view is a copy of the subjective which has gone through a process of *conditioning* - it may contain task, method or interrelationship deletions, modifications or insertions. The conditioned view is normally used for plan construction, so these modifications indirectly allow the problem solver performing the conditioning process to focus the attention of the scheduling and coordination mechanisms. As we will see below, the conditioned view can also represent the instantiation of the role assigned to it by the organizational structure.

2.2 Task and Goal Representation

Since the general purpose of TÆMS is to facilitate plan generation, it is well suited for representing the different task alternatives available to an agent in an organization. In an agent's subjective view we can represent (or dynamically generate) structures describing each of the high level goals the agent can achieve. Each of these structures would in turn describe the various alternate ways that a

particular goal might be achieved. The subjective view would then be, in this light, a complete description of all the possible roles an agent might be assigned to, and how the agent might act to satisfy that role.

Within a particular organizational structure, however, an agent will typically (but not necessarily) be working toward just a single or limited set of goals. Thus, in the conditioned view there will be a single task group representing that goal. The subtree underneath that task group might be further pruned to reflect other decisions within the organization. For example, in the subjective view there might be two possible ways to complete a task, one local solution and another using a remote contract, whereas an organizational constraint could remove the remote option from the conditioned view. So, using this representation we can encode all the tasks a particular agent might be working on, and also the specific task(s) they have chosen or been assigned. These techniques are used in the experiment shown in section 4 to control the path selection done by transporter agents.

2.3 Specifying Interactions

As mentioned above, TÆMS allows the agent to represent tasks and methods that other agents may perform. This capability allows TÆMS to model potential interactions between agents very effectively. Consider the case where agent C_1 requires resource X as part of its manufacturing process, as seen in Figure 3. Here, C_1 has a method Get-Materials, which consumes some amount of resource X. In the subjective view we can see that C_1 knows of three other agents that can produce X for it: P_1, P_2, and P_3, each of which is represented by a shaded, nonlocal method that has a produces interrelationship to X. In the conditioned view only P_2 is represented, which indicates that in this organization, C_1 should obtain X from P_2. A less restrictive organization might allow C_1 to choose probabilistically from either P_1 or P_2, which could be represented by adding P_1's produces interrelationship to the conditioned view. In this new model, the local scheduler would select from the two each time the resource is needed, based on the characteristics that differentiate the two produces interrelationships. This type of probabilistic usage relationship will be discussed further in the example in section 4.

Other interrelationship types might inform the agent that another agent's actions could enable, disable, facilitate or hinder local execution. Assuming the agent needs to interact, explicitly or not,

---

[1]There is also an omniscient *objective* view, inaccessible to agents, which defines the real execution alternatives. In simulation, one can engineer differences between the objective and subjective views to create scenarios where the agent's expectations are not met.
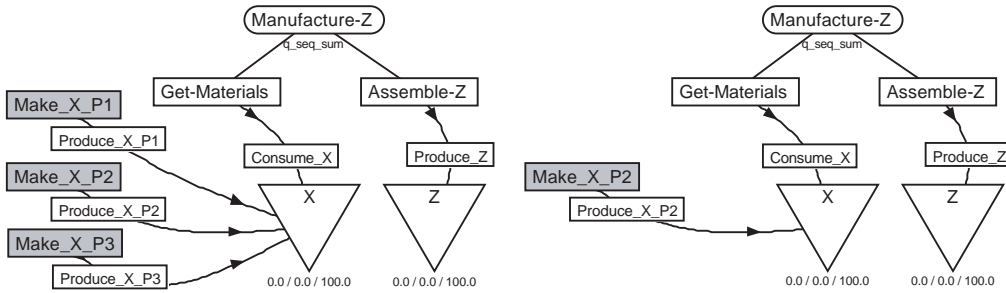
Figure 3: Subjective (left) and conditioned (right) views of C_1's task structure.

with those remote agents to exploit these interrelationships, they then indicate a point of potential coordination. For instance, if a remote agent's method enables one at the local agent (i.e. there is an enables interrelationship between them), the local agent must ensure that the remote method has successfully completed before it can succeed at its local method. This implies that some sort of co-ordination must take place to cause the correct ordering of events. Thus, an agent using this type of model can succinctly encode what sort of coordination is needed (based on the interrelationship type), with what other agents it should take place, and given a schedule of execution, when it should occur.

2.4 Other Organizational Details

Data concerning particular agents, existing commitments, and execution schedules are also stored within TÆMS models. Inevitably, however, there are some details particular to a given organization that do not directly fit into this representation. For these situations, all elements in a TÆMS model can be associated with an arbitrary set of attributes, where one could specify such things as preferred communication medium, optimal load measurements, or interaction history with a particular agent. Also stored here are performance characteristics, such as result thresholds and tolerances and expected frequency statistics, which the diagnosis subsystem can use to help identify potential failures.

With this information, we can now return to the questions posed in the overview section. The arrangement of agents can be expressed and derived locally by using the complete structure and owning agent tags of tasks and interrelationships. Commitments can exhibit potential influences on agent activities, or by explicitly modeling the task of obtaining goals from remote agents. Interrelationships can denote communication alternatives among agents, and their presence in the conditioned view determines if they should be explicit or implied. Our subjective view represents all the possible roles and responsibilities the agent may hold in the organization, while the conditioned view indicates its currently assigned position. The organizational search space is therefore specified by the range of possible conditioned structures. To adapt its role in the organization, the agent must develop an appropriate mapping from the subjective to conditioned. The TÆMS knowledge representation thus serves as a reasonable representation of the organizational structure; the task now is to use diagnosis to find the appropriate mapping.

3. THE DIAGNOSTIC SUBSYSTEM

Figure 4 shows the architecture of the diagnostics subsystem we currently employ. It uses a blackboard-based design, separating the process into three distinct layers: symptoms, diagnoses, and reactions. This type of system offers several advantages. It promotes a clear chain of reasoning, since the diagnoses supporting a given

reaction can easily be identified, as can the symptoms that support a particular diagnosis. Each layer is also subdivided by time, so a history of activity on each level is readily accessible. The blackboard layers also clearly define the separation of responsibilities. This modularity allows any of the layers to be accessed at any time, enabling arbitrary components or even remote agents to asynchronously use and add to elements on the blackboard. The different layers of the blackboard, and the components which make use of them (excepting the effect monitor), will be discussed below. In our current systems, each agent uses this subsystem to perform local diagnosis, although it is quite feasible that in other systems a specialized "diagnosis" agents would be responsible for monitoring small groups of their peers.

The lowest level of the blackboard contains symptoms, elements that contain observations about such things as the environment, agent activities and commitments. Two classes of components currently generate symptoms: *observers* and *modelers*. Observers work by simply monitoring different aspects of the agent, and generating symptoms when appropriate. Modelers take a more proactive approach by building or learning models, and then using these models as a basis for comparison, an approach similar to that used in conventional model-based diagnosis. As models are updated, or predictions derived from the models fail, appropriate symptoms describing these instances are noted on the blackboard. We have experimented with modelers that learn interrelationships in TÆMS objects [7] and others that predict environmental resource usage.

Diagnosis is a well-researched field, with many different methods and techniques already available to the system designer. Our goal was to use a technique that offered great flexibility in the information it could use and the diagnoses it could generate, without sacrificing subject scope or domain independence. It is not clear from the outset, however, that any single diagnostic technique (e.g., model-based, symptom-directed, collaborative) is suitable for the entire range of faults exhibited by multi-agent systems. It was therefore desirable to use a system or framework capable of incorporating different diagnostic techniques. In such an architecture we can make use of a variety of different methods, given the types of failures they best address, and the performance characteristics they exhibit (e.g. convergence time, scalability, efficiency, etc.).

Expanding on work first researched in [10], we chose to organize our diagnostic process using a causal model. The causal model is a directed, acyclic graph that organizes a set of diagnosis nodes. Figure 5 shows an example of such a graph; more examples of graphs addressing broader topics can be found in [1]. A more applied model used in the PCT domain can also be seen in Figure 6. Each node in the causal model corresponds to a particular diagnosis, with varying levels of precision and complexity. As a node produces a diagnosis, the causal model can determine what other, more detailed, diagnoses may further categorize the problem. Within the
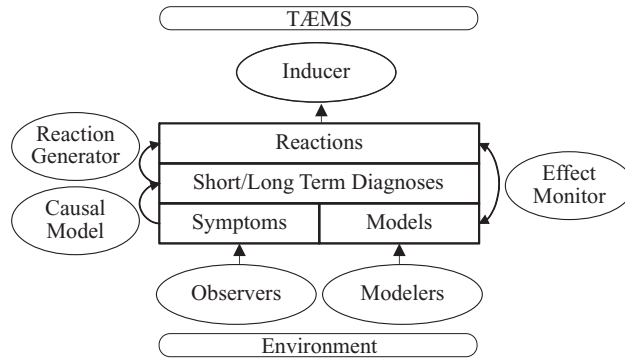
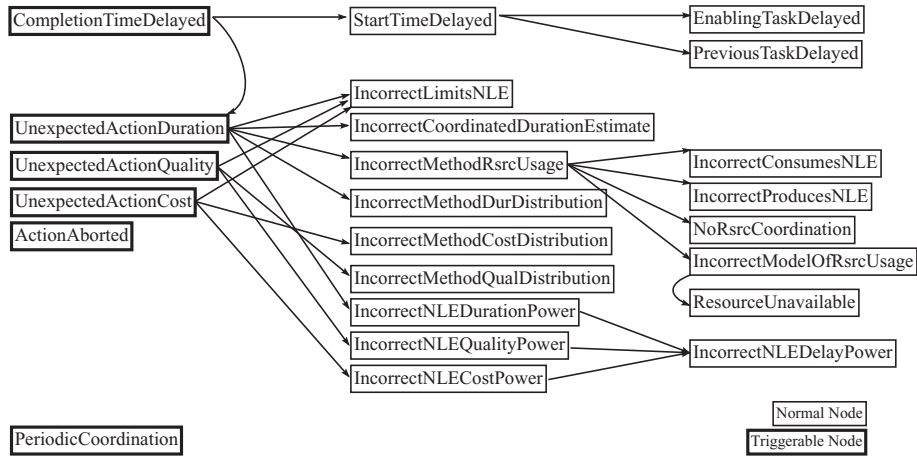**Figure 4: High-level architecture of the diagnostics subsystem.**

Figure 5: Causal model for diagnosing action- and coordination-based faults.

diagnosis system, the causal model then acts as a sort of road map, allowing diagnosis to progress from easily detectable symptoms to more precise diagnostic hypotheses as needed. A more advanced technique can also use the same structure to help validate diagnoses, by using backward chaining through the branches to determine the state of other potentially related diagnostic nodes.

It is worth mentioning that nodes in the causal model do not necessarily produce single-shot diagnoses. Some nodes, such as UnexpectedActionDuration, simply produce a diagnosis and stop. Others, such as PeriodicCoordination, can produce a diagnosis and monitor it over time to determine if conditions change or more evidence is found. Thus, a node could pose an initial diagnostic hypothesis when confronted with a particular situation. Since it only has limited evidence (presumably one data point), the confidence on that diagnosis would be low. The node can persist, however, and either passively watch for related evidence, or actively gather new information that either contradicts or corroborates the initial diagnosis. Furthermore, since other diagnoses or reactions may be based on that initial diagnosis, a change may also affect their confidence, causing a ripple effect throughout the blackboard as the original diagnosis accumulates new information.

The reactions level contains descriptions of the potential solutions to diagnoses found on the previous level. In some sense, then, these reactions are the effectors of organizational change. As diagnoses are hypothesized, and their confidence reaches a certain threshold, the reaction generator will pose solutions to those diagnoses. For instance, if the causal model determines that insufficient

resources were available for a particular action because their usage was not coordinated over, a potential reaction would modify the conditioned view of the agent's TÆMS model so that coordination would take place for that action in the future. A different reaction for that problem might remove the offending method from the view altogether. Similarly, if a diagnosis determined that an agent's actions were predictably periodic, a reaction could set up default commitments to reduce the need for explicit communication during each of those cycles. Like diagnoses, reactions can also be long-lived, providing incremental change in response to updated diagnoses or to slowly test new organizational changes.

Organizational changes for higher level characteristics work the same way. For instance, in Figure 3, a consumer's choice of producers limited is by the organization. A reaction could implement this change by removing the methods and interrelationships that describe those extra producers from the conditioned view. In the initial PCT organizations seen in Figure 1, a reaction would modify the conditioned view of F_3 to indicate it should use T_1. When this change is made, T_2 would be free to accept the transportation request from F_4.

Similar methods can drive more large scale reorganizations, although additional safeguards should be present to protect against the likely larger cost of failure. In these cases, local reactions can directly implement sophisticated reorganization techniques like those seen in [6, 8, 11, 9], or they can direct the local agent controller or problem solver to do so. For instance, local diagnosis could first determine that the control hierarchy for the current orga-

nization is inefficient or overwhelmed, because one or more high-level managing units was unable to cope with its workload. Because resolving such an issue can result in an interruption of service, this diagnosis would first cause a more detailed view of the situation to be analyzed, by evaluating different metrics, analyzing trends, or gathering additional evidence from remote sources. If this more advanced diagnosis also determines a problem exists, a reaction can be generated which prompts a more sophisticated direct, distributed search for a more appropriate organizational structure.

The task of selecting from among several potential reactions lies with the inducer. Our current inducer simply instantiates any reaction it sees on the blackboard. In future versions this component would be more complex, able to differentiate between reactions, analyze the potential benefits and drawbacks of each, and determine the best reaction given the agent's current context and prior history.

4. EXPERIMENTAL RESULTS

A specific system using the architecture outlined in the previous sections has been implemented and tested using scenarios from the PCT domain. In this section, we will outline one of those experiments, examining the effects of organizational changes in a small, eight member multi-agent system.

In this scenario, there are four factories and four transporters operating in the environment shown in Figure 1. As shown in that figure, there are also four "doorways", or potential points of contention along the lengthwise transporter routes. These doorways only allow one transporter through at a time, which transporters must be aware of as they select their routes. The objective for transporters is then to deliver their cargo on time, given the potential vagaries of factory production and the need to avoid collisions on travel pathways. Factories in the environment have different production capabilities and resource requirements, summarized in Table 1, and they must also select one or more transporters to deliver materials to them. Each factory is capable of producing both a simple resource, one that requires no external elements to build, and a complex resource, which requires other resources to produce. F_4 can also produce an even more complex resource Q, which is the combination of four other resources.

Factory	Simple	Complex
F_1	$\emptyset \rightarrow A$	$B + C \rightarrow X$
F_2	$\emptyset \rightarrow C$	$B + D \rightarrow Y$
F_3	$\emptyset \rightarrow D$	$A + C \rightarrow W$
F_4	$\emptyset \rightarrow B$	$A + D \rightarrow Z$
		$A + C + X + Y \rightarrow Q$

Table 1: Production rules for factories in PCT example.

In the initial phase of the scenario, the goal of each factory is to produce seven of each type of complex resource by time 700. After time 700, the objective shifts so that the system as a whole should produce as much Q as possible by time 1200. To provide further context, the round trip duration from F_1 to F_3 is around forty time units (barring path contention), and resource production can take five time units. Two organizational characteristics have alternatives as part of this scenario - the transporter selected for a particular transportation task, which is decided by the consuming factory, and the path the transporter selects to perform that task. Each consumer then chooses one of the four available transporters to satisfy its delivery needs. Each transporter has a choice of two different viable paths for any given delivery task. This latter selection is implemented probabilistically, so a given transporter might have a 70% chance of selecting path A, and a 30% of path B. Three

runs were performed, the first employed an arbitrary static organization, the second used diagnosis with only task allocation nodes from the causal model shown in Figure 6, and the third used the entire causal model, which added path selection diagnosis to the second trial. The objectives behind most of the nodes in the model should be intuitive: DeadlineMissed fires when action's deadline has not been achieved, TransporterOverloaded is true when a transporter's task load is disproportionate to those of it's peers, and OverCoordination determines when excessive coordination activity has been detected, as would occur when a particular route is highly contended. Organizational change occurs under two circumstances. A TransporterOverloaded diagnosis will result in the transporter attempting to shift some of its current delivery tasks to alternate transporters. This will also induce change in the consumer's organizational model such that future requests will go to the alternate transporter. An OverCoordination diagnosis associated with the route coordination protocol (which prevents transporters from colliding along a common path) will cause the transporter to adjust its local route selection probabilities.

Table 2 shows the results from the experiments; average delay is the average amount of time from when a factory begins gathering materials for resource production to when the resource is completed. As shown in the table, the results from the static organization are quite poor, except those for resource X, which benefitted from using relatively underloaded transporters in the organization. The delays during production of Q are particularly bad, being more than three times those in other runs. Clearly this performance is dependent on the organization that was initially chosen, but more important to this discussion is the fact that an initial poor organization was greatly improved with the addition of diagnostic-based adaptation, as shown by the results from the second and third runs.

Adaptation	Average Delay				
	W	X	Y	Z	Q
None	261.9	94.5	253.6	252.0	422.0
CM (Tasks)	97.8	88.4	90.3	98.4	118.9
CM (Full)	93.0	90.0	92.5	96.5	99.6

Table 2: Results from three trials in the PCT scenario.

In the results from the second run we see that the average production delay for each resource was reduced by nearly two-thirds in most cases. These gains were obtained by using load statistics, which can be generated from the transporter's conditional TÆMS view, to more efficiently allocate transporters to the various tasks available. Reallocation was performed by adding or removing interrelationships from a factory's conditional view, which constrained the set of transporters the factory could potentially use. Initially, the consumers chose from all transporters available in the environment, which was quite inefficient because transporters working on long-haul (diagonal) runs were selected as often as those on shorter runs. Through incremental changes to their conditional views, reacting to transporter performance and load, consumers in the second trial settled into an organization where more lightly-loaded transporters were selected more frequently, producing a more efficient allocation. The allocation for the initial phase settled around time 240, after four task reassignments. When the second phase started, after time 700, additional task reallocations took place every 100 pulses or so until the system completed.

With the introduction of path selection diagnosis in the third run the delays dropped again, especially that of resource Q, which due to its larger component set has the most potential for conflicting transporter routes. Diagnosis relevant to path selection was performed by the TooManyConflicts and OverCoordination nodes, which determined if a transporter encountered excessive conflicts
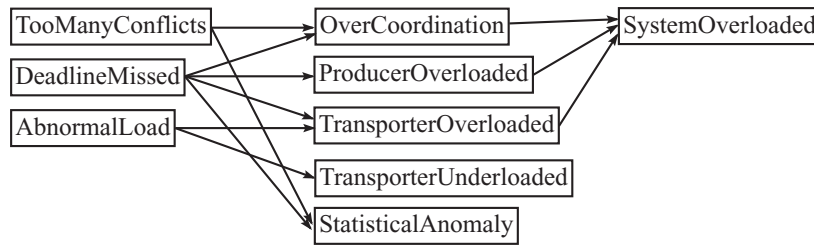
Figure 6: Causal model structure used in the PCT scenario.

when coordinating with other transporters over route usage. In this trial, the transporters' options regarding path selection were improved by constraining them in such a way to reduce the possibility of conflict. This was also implemented through local incremental change, this time by the transporters themselves, as they experimented with varying path probabilities (the chance that a particular route will be chosen) until one was found which incurred few conflicts. By lowering the potential for conflicts, the path probabilities reduced the overhead spent on both control decisions and coordination, which left more time for the actual act of transporting. Interestingly, despite similar final results, the organizational changes with both techniques available were very different than those of the previous trial. Periodic task reallocations were done every 100 pulses or so until time 800. Additionally, two to three path probability changes were made before time 200 for each of the transporters, and one or two more after time 700. These differences are caused by the fact that the adaptations were performed autonomously by individual agents in response to different efficiency metrics. Because no central authority governs the organizational changes, it is probable that the agents will adapt differently to the different metrics, but eventually settle on a similar result. Therefore, the various states the organization as a whole will go through towards this result will vary depending on the type of diagnosis being performed.

5. CONCLUSION

Generating an effective organizational structure for a multi-agent system is a crucial part of making them efficient, especially for large systems where global control is impractical. Adapting these organizations at runtime therefore becomes important when the environment, goals, or participants are liable to change. Several specific techniques have been offered by previous work in this area; we propose a more general solution to the problem by organizing such activity under the umbrella of diagnosis. A general diagnostic engine such as that shown in this paper is capable of detecting and diagnosing a variety of faults and inefficiencies, which can be used to drive organizational change. The organization itself is represented using models, such as TÆMS structures, which abstract the relevant portions of agents' capabilities and interactions in a way that facilitates both its use by agent control components and its adaptation by diagnosis. In this architecture, the methods driving change, and the characteristics affected by adaptation, can then be simplified to general techniques updating a domain independent representation, which can be reused from one system to the next.

A number of issues remain to be researched in this area. How efficient are the resulting organizations? How long does it take to discover a problem, and then to converge on a viable solution? Can one guarantee that the reactions will do no harm, and avoid oscillations? Our use of a blackboard structure, which can be searched for historical reactions, can help avoid these pitfalls. How much

of the diagnostic engine can be domain independent, and how are reaction values and thresholds calculated. Of these these latter issues, can some sort of learning technique be used to automate value selection? We hope to address these areas in future work.

6. REFERENCES

[1] Ana L.C. Bazzan, Victor Lesser, and Ping Xuan. Adapting an Organization Design through Domain-Independent Diagnosis. Comp Sci Technical Report TR-98-014, University of Massachusetts at Amherst, February 1998.

[2] Daniel D. Corkill and Victor R. Lesser. The use of meta-level control for coordination in a distributed problem solving network. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 748–755, Karlsruhe, Germany, August 1983.

[3] Keith S. Decker and Victor R. Lesser. Quantitative modeling of complex environments. *International Journal of Intelligent Systems in Accounting, Finance, and Management*, 2(4):215–234, December 1993. Special issue on "Mathematical and Computational Models of Organizations: Models and Characteristics of Agent Behavior".

[4] Edmund H. Durfee and Thomas A. Montgomery. Coordination as distributed search in a hierarchical behavior space. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1363–1378, 1991.

[5] Bryan Horling and Victor Lesser. Using diagnosis to learn contextual coordination rules. In *in Proceedings of the AAAI-99 Workshop on Reasoning in Context for AI Applications, a version also available as UMASS CS Tech Report TR99-15*. AAAI, 1999.

[6] Toru Ishida, Makoto Yokoo, and Les Gasser. An organizational approach to adaptive production systems. In *National Conference on Artificial Intelligence (AAAI-90)*, pages 52–58, 1990.

[7] D. Jensen, M. Atighetchi, R. Vincent, and V. Lesser. Learning quantitative knowledge for multiagent coordination. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, Orlando, FL, July 1999. AAAI.

[8] Onn Shehory, Katia Sycara, Prasad Chalasani, and Somesh Jha. Agent cloning: An approach to agent mobility and resource allocation. *IEEE Communications*, 36(7):58–67, July 1998.

[9] Y. So and E. H. Durfee. Modeling and designing computational organizations. In *Working Notes of the AAAI Spring Symposium on Computational Organization Design*, 1994.

[10] T. Sugawara and V. Lesser. Learning control rules for coordination. In *Multi-Agent and Cooperative Computation '93*, pages 121–136, 1993.

[11] Roy Turner and Elise Turner. Organization and reorganization of autonomous oceanographic sampling networks. In *Proceedings of the IEEE Interlational Conference on Robotics and Automation*, 1998.

[12] Thomas Wagner and Victor Lesser. Relating quantified motivations for organizationally situated agents. In N.R. Jennings and Y. Lespérance, editors, *Intelligent Agents VI — Proceedings of the Sixth International Workshop on Agent Theories, Architectures, and Languages (ATAL-99)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2000.

An Agent Infrastructure to Build and Evaluate Multi-Agent Systems: The Java Agent Framework and Multi-Agent System Simulator *

Regis Vincent, Bryan Horling, and Victor Lesser

Dept of Computer Science,
University of Massachusetts,
Amherst MA 01003
USA
{vincent,bhorling,lesser}@cs.umass.edu

Abstract. In this paper, we describe our agent framework and address the issues we have encountered designing a suitable environmental space for evaluating the coordination and adaptive qualities of multi-agent systems. Our research direction is to develop a framework allowing us to build different type of agents rapidly, and to facilitate the addition of new technology. The underlying technology of our Java Agent Framework (JAF) uses a component-based design. We will present in this paper, the reasons and the design choices we made to build a complete system to evaluate the coordination and adaptive qualities of multi-agent systems.
Abbreviation:
- JAF Java Agent Framework;
- MASS Multi-Agent System Simulator

1   Introduction

Agent technology, in one form or another, is gradually finding its way into mainstream computing use, and has the potential to improve performance in a wide range of computing tasks. While the typical commercial meaning of the word agent can refer to most any piece of software, we believe the real potential of this

---

paradigm lies with more sophisticated, autonomous entities. In general, our definition of an *agent* is an autonomous entity capable of reacting to its environment, determining its most appropriate goals and actions in its world, and reasoning about deadlines and tradeoffs arising from those determinations. To correctly develop such autonomous, intelligent, reactive pieces of software, we must have good ways of implementing, debugging and evaluating them. Many researchers have realized this, and have begun to develop the required infrastructure [2, 10, 16, 20, 3, 19]. Our research has done the same, but with a different approach. Our direction is to develop a framework allowing us to build different type of agents rapidly, and to facilitate the addition of new technology.

The underlying technology of our Java Agent Framework (JAF) uses a component-based design. Developers can use this plug and play interface to build agents quickly using existing generic components, or to develop new ones. For instance, a developer may require planning, scheduling and communication services in their agent. Generic scheduling and communication components exist, but a domain-dependent planning component is needed. Additionally, the scheduling component does not satisfy all the developer's needs. Our solution provides the developer with the necessary infrastructure to create a new planning component, allowing it to interact with existing components without unduly limiting its design. The scheduling component can be derived to implement the specialized needs of their technology, and the communication component can be used directly. All three can interact with one another, maximizing code reuse and speeding up the development process. We also respect the fact that researchers require flexibility in the construction of their software, so in general, our solution serves as simple scaffolding, leaving the implementation to the developer beyond a few API conventions.

Much of the generality available in existing JAF components is derived from their common use of a powerful, domain-independent representation of how agents can satisfy different goals. This representation, called TÆMS [4, 5], allows complex interactions to be phrased in a common language, allowing individual components to interact without having direct knowledge of how other components function. Implemented components in JAF are designed to operate with relative autonomy. Coincidentally, a reasonable analogy for a JAF agent's internal organization is a multi-agent system, to the degree that each has a limited form of autonomy, and is capable of interacting with other components in a variety of ways. They are not sophisticated agents, but within the agent, individual components do provide specific, discrete functionality, and may also have fixed or dynamic goals they try to achieve. This functionality can be requested by components via direct method invocation, or it may be performed automatically in response to messages or events occurring in the agent.

Our objective was to allow developers to implement and evaluate systems quickly without excessive knowledge engineering. This way, one can avoid working with domain details, leaving more time and energy to put towards the more critical higher level design. We have also focused on more precise and controlled methods of agent evaluation technologies. Together with the agent framework, we

24

have built a simulation environment for the agents to operate in. The motivation for the Multi-Agent System Simulator (MASS) is based on two simple, but potentially conflicting, objectives. First, we must accurately measure and compare the influence of different multi-agent strategies in an deterministic environment. At the same time, it is difficult to model adaptive behavior realistically in multi-agent systems within a static environment, for the very reason that adaptivity may not be fully tested in an environment that does not substantively change. These two seemingly contradictory goals lie at the heart of the design of MASS - we must work towards a solution that leads to reproducible results, without sacrificing the dynamism in the environment the agents are expected to respond to.

In this paper, we describe our agent framework and address the issues we have encountered designing a suitable environmental space for evaluating the coordination and adaptive qualities of multi-agent systems. In the following sections, we will describe both the JAF framework and the MASS simulation environment. To describe how these concepts work in practice, we will also present an example implemented system, the Intelligent Home (IHome) domain testbed. Lastly, we present an example of the how a JAF-based multi-agent system can run in an alternate simulated environment, and also how it was migrated to a real-time, hardware-based system. We conclude with a brief overview of the future directions of this project.

2 Java Agent Framework

An architecture was needed for the agents working within the MASS environment which effectively isolated the agent-dependent behavior logic from the underlying support code which would be common to all of the agents in the simulation. One goal of the framework was therefore to allow an agent's behavioral logic to perform without the knowledge that it was operating under simulated conditions, e.g. a problem solving component in a simulated agent would be the same as in a real agent of the same type. This clean separation both facilitates the creation of agents, and also provides a clear path for migrating developed technologies into agents working in the real world. As will be shown later, this has been recently done in a distributed sensor network environment, where agents were migrated from a simulated world to operating on real hardware [15]. The framework also needed to be flexible and extensible, and yet maintain separation between mutually dependent functional areas to the extent that one could be replaced without modifying the other. To satisfy these requirements, a component-based design, the Java Agent Framework (JAF) [12], was created[1].

Component based architectures are relatively new arrivals in software engineering which build upon the notion of object-oriented design. They attempt to encapsulate the functionality of an object while respecting interface conventions,

---

[1] This architecture should not be confused with Sun's agent framework of the same name.

thereby enabling the creation of stand alone applications by simply plugging together groups of components. This paradigm is ideal for our agent framework, because it permits the creation of a number of common-use components, which other domain-dependent components can easily make use of in a plug-and-play manner. Note that the agents produced with this scheme act as small multi-agent systems in and of themselves, where components function as partially autonomous entities that communicate and interact to achieve their individual goals. For instance, our system has a scheduling component, whose goal it is to schedule activities as best as possible, respecting quality, time and resource constraints. It can operate in several ways, the most common being to respond to events describing new tasks needing to be performed. On receiving such an event, the scheduler attempts to integrate these actions into the existing schedule, which in turn will be used by an execution component to determine when to perform the actions. Thus, the scheduling component operates autonomously, reacting to changes and requests induced by other components. This arrangement is key to the flexibility of JAF. Because other components for the most part do not care how or where in an agent an operation is performed, the designer is free to add, modify or adapt components as needed.
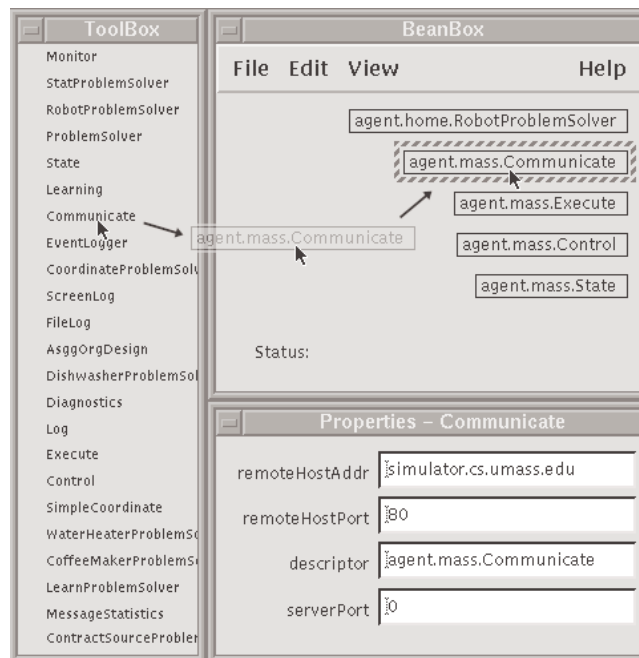

Fig. 1. Sun Beanbox, which can be used to build JAF agents.

JAF is based on Java Beans, Sun Microsystem's component model architecture. Java Beans supplies JAF with a set of design conventions, which provides behavior and naming specifications that every component must adhere to. Specifically, the Java Beans API gives JAF a set of method naming and functional conventions which allow both application construction tools and other beans to manipulate a component's state and make use of its functionality easily. This is important because it provides compatibility with existing Java Beans tools, and facilitates the development process by providing a common implementation style among the available components. JAF also makes heavy use of Java Bean's notion of event streams, which permit dynamic interconnections to form between stream generating and subscribing components. For instance, we have developed a causal-model based diagnosis component [13] which tracks the overall performance of the agent, and makes suggestions on how to optimize or repair processes performed by, or related to, the agent. The observation and diagnosis phase of this technology is enabled by the use of dynamic event streams, which the diagnosis component will form with other components resident in the agent. The component will begin by listening to one or more components in the agent, such as the local coordination component. This stream could tell the diagnosis component when coordination attempts where made, who the remote agents were, whether the coordination succeeded or not, and if the resulting commitment was respected. Events arising from this component are analyzed, and used to discover anomalous conditions. In the case of the coordination component, a series of similar failed coordination attempts could indicate that a particular remote agent has failed, or that it no longer provides the desired service. More proactive analysis into the current state of the coordination component could then yield further information. By both monitoring the events the components produce, and the state they are currently in, the diagnosis component can determine if the components are performing correctly, and generate potential solutions to the problems it finds. Our experience with the diagnosis component was that we did not have to modify other components in order to integrate its functionality.
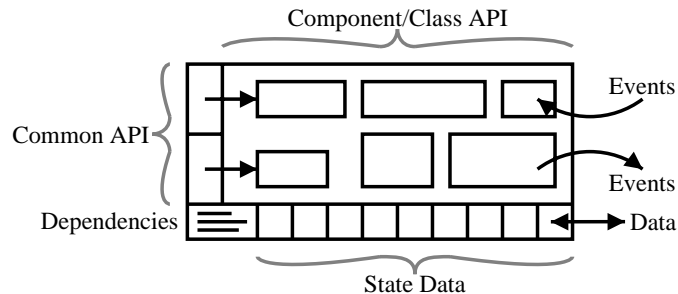

Fig. 2. Abstract view of a typical JAF component.

JAF builds upon the Java Beans model by supplying a number of facilities designed to make component development and agent construction simpler and more consistent. A schematic diagram for a typical JAF component can be seen in figure 2. As in Java Beans, events and state data play an important role in some types of interactions among components. Additional mechanisms are provided in JAF to specify and resolve both data and inter-component dependencies. These methods allow a component, for instance, to specify that it can make use of a certain kind of data if it is available, or that it is dependent on the presence of one or more other components in the agent to work correctly. A communications component, for example, might specify that it requires a local network port number to bind to, and that it requires a logging component to function correctly. These mechanisms were added to organize the assumptions made behind flexible autonomy mentioned above - without such specifications it would be difficult for the designer to know which services a given component needs to be available to function correctly. More structure has also been added to the execution of components by breaking runtime into distinct intervals (e.g. initialization, execution, etc.), implemented as a common API among components, with associated behavioral conventions during these intervals. Individual components will of course have their own, specialized API, and "class" APIs will exist for families of components. For instance a family of communication components might exist, each providing different types of service, while conforming to a single class API that allows them to easily replace one another.

The goal of a designer using JAF is to use and add to a common pool of components. Components from this pool are combined to create an agent with the desired capabilities (see Figure 1). For instance, rather than regenerating network messaging services for each new project, a single Communicate component from the pool can be used from one domain to the next. This has the added benefit that once a component has been created, it may be easily swapped out of each agent with one that respects the original class API, but offers different services. Later in this paper we will describe the MASS simulation environment, which provides simulation and communication services to agents. Messages sent from an agent working in this environment must be routed through MASS, which requires a specialized Communication component which is "aware" of MASS and how to interact with it. In our pool of components we thus have a simple Communicate which operates in the conventional sense using TCP, and a MASS Communicate which automatically routes all messages through the simulation controller. Components using communication need not be aware of the internal delivery system being used, and can therefore be used without modification in both scenarios. Revisiting the hybrid simulation issue raised earlier, we can have an agent which conforms to the MASS communication specification, or uses real world messaging as needed by just exchanging these two components. Analogously, one could have a MASS Execute component, which uses the simulator to perform all executable actions, or one which actually performed some actions locally, and reported the results to the MASS controller when completed. In this

latter case, the consistency of the simulation environment is maintained through the notification, but real data may be still generated by the agent.

The organization of a JAF agent does not come without its price. The autonomous nature of individual components can make it difficult to trace the thread of control during execution, a characteristic exacerbated by the use of events causing indirect effects. It can also be difficult to implement new functionalities in base components, while respecting conventions and APIs in derived ones. However, we feel the flexibility, autonomy and encapsulation offered by a component oriented design makes up for the additional complexity.

To date, more than 30 JAF components have been built. A few of these are explained below.

- **Communicate** This component serves as the communication hub for the agent. TCP based communication is provided through a simple interface, for sending messages of different encodings (KQML, delimited or length-prefixed). It also serves as both a message receiver and connection acceptor. Components interact with Communicate by listening for message events, or by directly invoking a method to send messages. Derived versions exist to work with MASS and other simulation environments.

- **Preprocess Taems Reader** TÆMS is our task description language, which will be covered later in this article. This component allows the agent to maintain a library of TÆMS structure templates, which can be dynamically instantiated in different forms, depending on the needs of the agent. For example, the designer may update method distributions based on learned knowledge, or add in previously unrecognized interactions as they are discovered. This is important because it facilitates the problem solving task by allowing the developer to condition generated task structures with respect to current working conditions. Data manipulation capabilities exist which permit mathematical and conditional operations, along with TÆMS structure creation and manipulation. Simple routines can then be written with these tools to use information given to the preprocessor to condition the structure. The ability to perform these operations within the TÆMS file itself allows the problem solving component to be more generic. A derived version of the component also exists which reads simple static task structure descriptions.

- **Scheduler** The scheduling component, based on our Design-To-Criteria (DTC) scheduling technology [22], is used by other components to schedule the TÆMS task structures mentioned above. The resulting schedule takes into account the cost and quality of the alternative solutions, and their durations relative to potential deadlines. The scheduler functions by both monitoring state for the addition of new TÆMS structures, for which it will produce schedules, and through direct invocation.

- **Partial Order Scheduler** A derived version of the Scheduler component, the partial-order scheduler provides the agent with a more sophisticated way of managing its time and resources [21]. Replacing the Scheduler with this component allows the agent to correctly merge schedules from different structures, exploit areas of potential parallelism, and make efficient use

of available resources. Functionally, it provides a layer on top of the DTC Scheduler component, first obtaining a conventional schedule as seen above. It then uses this to reason about agent activity in a partially-ordered way - concentrating on dependencies between actions and resources, rather then just specifying times when they may be performed. This characteristic allows agents using the partial order scheduler to more intelligently reason about when actions can and can not be performed, as well as frequently speeding up failure recovery by avoiding the need to replan.

– **State** The state component serves as an important indirect form of inter-action between components by serving as a local repository for arbitrary data. Components creating or using common information use State as the medium of exchange. Components add data through direct method calls, and are notified of changes through event streams. Thus one component can react to the actions of another by monitoring the data that it produces. For instance, when the problem solving component generates its task and places it in State, the scheduler can react by producing a schedule. This sched-ule, also placed in State, can later be used by the execution component to perform the specified actions.

– **Directory Services** This component provides generic directory services to local and remote agents. The directory stores multi-part data structures, each with one or more keyed data fields, which can be queried through boolean or arithmetic expressions. Components use directory services by posting queries to one or more local or remote directories. The component serves as an intermediary for both the query and response process, monitor-ing for responses and notifying components as they arrive. This component can serve as the foundation to a wide variety of directory paradigms (e.g. yellow pages, blackboard, broker).

– **FSM Controller** The FSM component can be used as a common interface for messaging protocols, specifically for coordination and negotiation inter-actions. It is first used to create a finite state machine describing the protocol itself, including the message types, when they can arrive, and what states a particular message type should transition the machine to. This scaffold-ing, provided by the FSM and used by the FSM Controller at runtime, is then populated by the developer with code to send and process the differ-ent messages. This clean separation between a protocol and its usage allows protocols to be quickly migrated from one environment to the next.

Other components provide services for logging, execution, local observation, diagnosis, and resource modeling, as well as more domain dependent functions. Examples of agents implemented with JAF will be covered later in this article.

3 Evaluation Environment for Multi-Agent Systems

Numerous problems arise when systematic analysis of different algorithms and techniques needs to be performed. If one works with a real-world MAS, is it pos-sible to know for certain that the runtime environment is identical from one run

to the next? Can one know that a failure occurs at exactly the same time in two different runs when comparing system behavior? Can it be guaranteed that inter-agent message traffic will not be delayed, corrupted, or non-deterministically interleaved by network events external to the scenario?

If one works within a simulated environment, how can it be known that the system being tested will react optimally a majority of the time? How many different scenarios can be attempted? Is the number is large enough to be representative?

Based on these observations, we have tried to design an environment that allows us to directly control the baseline simulated environment (e.g. be deterministic from one run to the next) while permitting the addition of "deterministically random" events that can affect the environment throughout the run. This enables the determinism required for accurate coordination strategy comparisons without sacrificing the capricious qualities needed to fully test adaptability in an environment.

Hanks et al. define in [11] several characteristics that multi-agent system simulators should have:

- **Exogenous events**, these allow exogenous or unplanned events to occur during simulation.
- **Real-world complexity** is needed to have a realistic simulation. If possible, the simulated world should react in accordance with measures made in the real world. Simulated network behavior, for instance, may be based on actual network performance measures.
- **Quality and cost of sensing and effecting** needs to be explicitly represented in the test-bed to accurately model imperfect sensors and activators. A good simulator should have a clear interface allowing agents to "sense" the world.
- **Measures of plan quality** are used by agents to determine if they are going to achieve their goal, but should not be of direct concern to the simulator.
- **Multiple agents** must be present to simulate inter-agent dependencies, interactions and communication. A simulator allowing multiple agents increases both its complexity and usefulness by adding the ability to model other scenarios, such as faulty communications or misunderstanding between agents, delay in message transfer.
- **A clean interface** is at the heart of every good simulator. We go further than this by claiming that the agents and simulator should run in separate processes. The communication between agents and simulator should not make any assumptions based on local configurations, such as shared memory or file systems.
- **A well defined model of time** is necessary for a deterministic simulator. Each occurring event can be contained by one or more points in time in the simulation, which may be unrelated to real-world time.
- **Experimentation** should be performed to stress the agents in different classes of scenarios. We will also add **deterministic experimentation** as another important feature of a simulator. To accurately compare the results separate runs, one must be sure that the experimental parameters are those which produce different outcomes.

We will show in this section how MASS addresses these needs. One other characteristic, somewhat uncommon in simulation environments, is the ability

to have agents perform a mixture of both real and simulated activities. For instance, an agent could use the simulation environment to perform some of its actions, while actually performing others. Executable methods, sensor utilization, spatial constraints and even physical manifestations fall into this category of activities which an agent might actually perform or have simulated as needed. An environment offering this hybrid existence offers two important advantages: more realistic working conditions and results, and a clear path towards migrating work from the laboratory to the real world. We will revisit how this can be implemented in later sections.
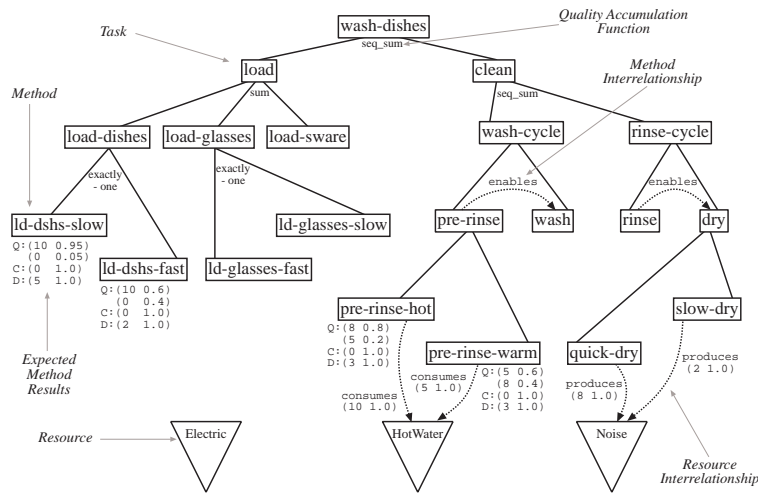

Fig. 3. TÆMS task structure for the IHome Dishwasher agent

4   Multi Agent System Simulator

MASS is a more advanced incarnation of the TÆMS simulator created by Decker and Lesser in 1993 [7]. It provides a more realistic environment by adding support for resources and resource interactions, a more sophisticated communication model, and mixed real and simulated activity. It also adds a scripting language, a richer event model, and a graph-like notion of locations and connectors in which agents can move about (e.g. rooms and doorways, or towns and roads). The new MASS simulator is completely domain independent; all domain knowledge is obtained either from configuration files or data received from agents working in the environment.

Agents running in the MASS environment use TÆMS [14, 6], a domain-independent, hierarchical representation of an agent's goals and capabilities (see Figure 3), to represent their knowledge. TÆMS , the Task Analysis, Environmental Modeling and Simulation language, is used to quantitatively describe the alternative

ways a goal can be achieved [9, 14]. A TÆMS task structure is essentially an annotated task decomposition tree. The highest level nodes in the tree, called task groups, represent goals that an agent may try to achieve. The goal of the structure shown in figure 3 is `wash-dishes`. Below a task group there will be a set of tasks and methods which describe how that task group may be performed, including sequencing information over subtasks, data flow relationships and mandatory versus optional tasks. Tasks represent sub-goals, which can be further decomposed in the same manner. `clean`, for instance, can be performed by completing `wash-cycle`, and `rinse-cycle`. Methods, on the other hand, are terminal, and represent the primitive actions an agent can perform. Methods are quantitatively described, in terms of their expected quality, cost and duration. `pre-rinse-warm`, then, would be described with its expected duration and quality, allowing the scheduling and planning processes to reason about the effects of selecting this method for execution. The quality accumulation functions (QAF) below a task describes how the quality of its subtasks is combined to calculate the task's quality. For example, the `sum` QAF below `load` specifies that the quality of `load` will be the total quality of all its subtasks - so only one of the subtasks must be successfully performed for the `sum` task to succeed. Interactions between methods, tasks, and affected resources are also quantitatively described as interrelationships. The `enables` between `pre-rinse` and `wash`, for instance, tells us that these two must be performed in order. The curved lines at the bottom of figure 3 represent resource interactions, describing, for instance, the different consumes effects method `pre-rinse-hot` and `pre-rinse-warm` has on the resource `HotWater`.

One can view a TÆMS structure as a prototype, or blueprint, for a more conventional domain-dependent problem solving component. In lieu of generating such a component for each domain we apply our technologies to, we use a domain independent component capable of reasoning about TÆMS structures. This component recognizes, for instance, that interrelationships between methods and resources offer potential areas for coordination and negotiation. It can use the quantitative description of method performance, and the QAFs below tasks, to reason about the tradeoffs of different problem solving strategies. The task structure in figure 3 was used in this way to implement the washing machine agent for the intelligent home project discussed later in this article. Figure 6 shows how an agent in the distributed sensor network domain (also discussed later), can initialize its local sensor. With this type of framework, we are essentially able to abstract much of the domain-dependence into the TÆMS structure, which reduces the need for knowledge engineering, makes the support code more generic, and allows research to focus on more intellectual issues.

Different *views* of a TÆMS structure are used to cleanly decouple agents from the simulator. A given agent will make use of a *subjective* view of its structure, a local version describing the agent's beliefs. MASS, however, will use an *objective* view, which describes the true model of how the goals and actions in the structure would function and interact in the environment. Differences engineered between these two structures allow the developer to quickly generate and test situations

where the agent has incorrect beliefs. We will demonstrate below how these differences can be manifested, and what effects they have on agent behavior. This technique, coupled with a simple configuration mechanism and robust logging tools, make MASS a good platform for rapid prototyping and evaluation of multi-agent systems.

The connection between MASS and JAF is at once both strong and weak. A JAF agent running within a MASS environment uses the simulator for the vast majority of its communication and execution needs, by employing "MASS-aware" components which route their respective data and requests through the simulation controller. The agent also provides the simulator with the objective view of its task structure, as well as the resources it provides and its location. The simulator in turn gives the agent a notion of time, and provides more technical information such as a random number generator seed and a unique id. Despite this high level of interconnection, the aspects of a JAF agent performing these actions are well-encapsulated and easily removed. Thus, an agent can be run outside of MASS by simply replacing those MASS-aware components with more generic ones. Outside of MASS, an agent would use conventional TCP/IP based communication, and would perform its actions locally. It would, for instance, use the local computer's clock to support its timeline, and read the random seed from a configuration file. An example of how this type of separation can be achieved will be covered in section 5.2, where we will show JAF agents running both in a different simulation environment, and independently on real hardware.
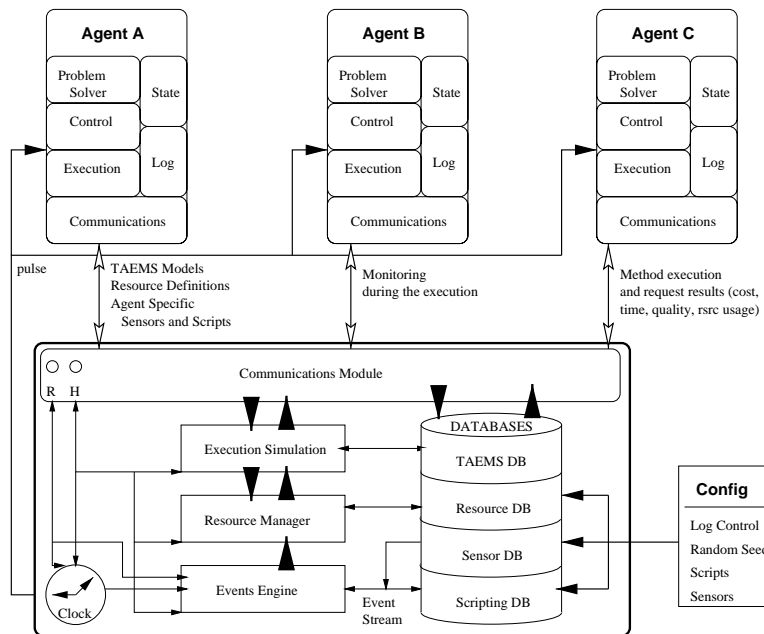

Fig. 4. Architecture of the MASS and agent systems.

Figure 4 shows the overall design of MASS, and, at a high level, how it interacts with the agents connected to it. On initialization, MASS reads in its configuration, which defines the logging parameters, random seed, scripts (if any) and global sensor definitions. These are used to instantiate various subsystems and databases. While the simulator itself is not distributed, connections to the simulator are made with standard TCP-based sockets, so agents can be distributed among remote systems. When connected, agents will send additional information to be incorporated into this configuration, which allows environmental characteristics specific to the agent to be bundled with that agent. For instance, an agent might send a description of a sensor which it needs to function, or a resource which it makes available to the environment. Arguably the most important piece of data arising from the agents is their TÆMS task structures, which are assimilated into the TÆMS database shown in figure 4. This database will be used by the execution subsystem during simulation to quantify both the characteristics of method execution and the effects resource and method interactions have on that method. The resource manager is responsible for tracking the state of all resources in the environment, and the event engine manages the queue of events which represent tangible actions that are taking place. The last component shown here, the communications module, maintains a TCP stream connection with each agent, and is responsible for routing the different kinds of messages between each the agent and their correct destination within the controller.

The MASS controller has several tasks to perform while managing simulation. These include routing message traffic to the correct destination, providing hooks allowing agents to sense the virtual environment and managing the different resources utilized by the agents. Its primary role, however, is to simulate the execution of methods requested by the agents. Each agent makes use of its partial, subjective view of the environment, typically describing its local view of a goal and possible solutions, which determines the expected values resulting from such an execution. As mentioned above, the simulator also has the true, objective view of the world which it uses to compute the results of activities in the environment. The distributions from the objective view are used when computing the values for a method execution, and for determining the results of method or resource interactions. This probabilistic distribution describes the average case outcomes; the simulator will degrade or improve results as necessary if, for instance, required resources are not available, or other actions in the environment enable or facilitate the method's execution in some way. For example, consider what would happen if the `enables` interrelationship between `rinse` and `dry` were absent in the subjective view of figure 3. During scheduling, the agent would be unaware of this interdependency, and thus would not enforce an ordering constraint between the two actions. If the agent were to perform `dry` first, the simulator would detect that its precondition `rinse` had not been performed, and would report that the `dry` method failed. In this case, the agent would need to detect and resolve the failure, potentially updating its subjective view with more accurate information.

MASS is also responsible for tracking the state and effects of resources in the environment. Figure 3 shows three such resources: `Electricity`, `HotWater`, and `Noise`. Two types of resources are supported - consumable and non-consumable. The level of a consumable resource, like `HotWater` is affected only through direct consumption or production. A non-consumable resource, like `Noise`, has a default level, which it reverts back to whenever it is not being directly modified. MASS uses the objective view from each agent to determine the effects a given method will have on the available resources. Also present in the objective view is a notion of bounds, both upper and lower, which the resource's level cannot exceed. If an agent attempts to pass these bounds, the resource switches to an error state, which can affect the quality, cost and duration of any action currently using that resource. At any given time, MASS must therefore determine which methods are affecting which resources, what effects those actions will have on the resources' levels, if the resource bounds have been exceeded, and what quantitative repercussions there might be for those violations.

Another responsibility consuming a large portion of the simulator's attention is to act as a message router for the agents. The agents send and receive their messages via the simulator, which allows the simulation designer to model adverse network conditions through unpredictable delays and transfer failures. This routing also plays an important role in the environment's general determinism, as it permits control over the order of message receipt from one run to the next. Section 4.1 will describe this mechanism in more detail.

4.1 Controllable Simulation

In our simulated experiments, our overriding goal is to be able to compare the behavior of different algorithms in the same environment under the same conditions. To correctly and deterministically replicate running conditions in a series of experiments, the simulator should have its own notion of time, "randomness" and sequence of events. Two simulation techniques exist which we have exploited to achieve this behavior: discrete time and events. Discrete time simulation segments the environmental time line into a number of slices. In this model, the simulator begins a time slice by sending a pulse to all of the actors involved, which allows them to run for some period of (real) CPU time. In our model, a pulse does not represent a predefined quantity of CPU time, instead, each agent decides independently when to stop running. This allows agent performance to remain independent of the hardware it runs on, and also allows us to control the performance of the technique itself. To simulate a more efficient scheduling algorithm, for instance, one could simply reduce the number of pulses required for it to complete. Since the agent dictates when it is finished its work, this can be easily accomplished by performing more work before the response is sent. This allows us to evaluate the potential effects of code optimization before actually doing it. The second characteristic of this simulation environment is its usage of events, which are used to instigate reactions and behaviors in the agent. The MASS simulator combines these techniques by dividing time into a number of slices, during which events are used to internally represent actions and

interact with the agents. In this model, agents then execute within discrete time slices, but are also notified of activity (method execution, message delivery, etc.) through event notification.

In the next section we will discuss discrete time simulation and the benefits that arise from using it. We will then describe the need for an event based simulation within a multi-agent environment.

Discrete time simulation Because MASS utilizes a discrete notion of time, all agents running in the environment must be synchronized with the simulator's time. To enable this synchronization, the simulator begins each time slice by sending each agent a "pulse" message. This pulse tells the agent it can resume local execution, so in a sense the agent functions by transforming the pulse to some amount of real CPU time on its local processor. This local activity can take an arbitrary amount of real time, up to several minutes if the action involves complex planning, but with respect to the simulator, and in the perceptions of other agents, it will take only one pulse. This technique has several advantages:

1. A series of actions will always require the same number of pulses, and thus will always be performed in the same amount of simulation time. The number of pulses is completely independent of where the action takes place, so performance will be independent of processor speed, available memory, competing processes, etc...
2. Events and execution requests will always take place at the same time. Note that this technique does not guarantee the ordering of these events within the time slice, which will be discussed later in this section.

Using this technique, we are able to control and reproduce the simulation to the granularity of the time pulse. Within the span of a single pulse however, many events may occur, the ordering of which can affect simulation results. Messages exchanged by agents arrive at the simulator and are converted to events to facilitate control over how they are routed to their final destination. Just about everything coming from the agents, in fact, is converted to events; in the next section we will discuss how this is implemented and the advantages of using such a method.

Event based simulation *Events* within our simulation environment are defined as actions which have a specific starting time and duration, and may be incrementally realized and inspected (with respect to our deterministic time line, of course). Note that this is different from the notion of event as it is traditionally known in the simulation community, and is separate from the notion of the "event streams" which are used internally to the agents in our environment.

All of the message traffic in the simulation environment is routed through the simulator, where it is instantiated as a message event. Similarly, execution results, resource modifiers or scripted actions are also represented as events within the simulation controller. We attempt to represent all activities as events both

for consistency reasons and because of the ease with which such a representation can be monitored and controlled.

The most important classes of events in the simulator are the *execution* and *message* events. An *execution* event is created each time an agent uses the simulator to model a method's execution. As with all events, execution events will define the method's start time, usually immediately, and duration, which depends on the method's probabilistic distribution as specified in the objective TÆMS task structure (see section 3). The execution event will also calculate the other qualities associated with a method's execution, such as its cost, quality and resource usage. After being created, the execution event is inserted into the simulator's time based event queue, where it will be represented in each of the time slots during which it exists. At the point of insertion, the simulator has computed, but not assigned, the expected final quality, cost, duration and resource usage for the method's execution. These characteristics will be accrued (or reduced) incrementally as the action is performed, as long as no other events perturbate the system. Such perturbations can occur during the execution when forces outside of the method affect its outcome, such as a limiting resource or interaction with another execution method. For example, if during this method's execution, another executing method overloads a resource required by the first execution, the performance of the first will be degraded. The simulator models this interaction by creating a limiting event, which can change one or more of the performance vectors of the execution (cost, quality, duration) as needed. The exact representation of this change is also defined in the simulator's objective TÆMS structure.

As an example, we can trace the lifetime of an action event in the MASS system - the `pre-rinse-hot` method from figure 3. The action begins after the agent has scheduled and executed the action, which will typically be derived from a TÆMS task structure like that seen in the figure. The Execute component in the agent will redirect this action to MASS, in the form of a network message describing the particular method to be executed. MASS will then resolve this description with its local objective TÆMS structure, which will contain the true quantitative performance distributions of the method. When found, it will use these distributions to determine the resulting quality, cost and duration of the method in question, as well as any resource effects. In this case, MASS determines the results of `pre-rinse-hot` will have a quality of 8, a cost of 0 and a duration of 3. In addition, it also determines the method will consume 10 units of `HotWater` for each time unit it is active. An action event is created with these values, and inserted into MASS's action queue. Under normal conditions, this event will remain in the queue until its assigned finish time arrives, at which point the results will be sent to the agent. Interactions with other events in the system, however, can modify the result characteristics. For instance, if the `HotWater` resource becomes depleted during execution, or if a conflicting method is invoked, the duration of the action may be extended, or the quality reduced. These effects may change the performance of the action, and thus may change the results reported to the agent.

Real activities may be also incorporated into the MASS environment by allowing agents to notify the controller when it has performed some activity. In general, methods are not performed by MASS so much as they are approximated, by simulating what the resulting quality, cost and duration of the action might be. Interactions are simulated among methods and resources, but only in an abstract sense, by modifying those same result characteristics of the target action. The mechanism provided by MASS allows for mixed behavior. Some actions may be simulated, while others are performed by the agent itself, producing the actual data, resources or results. When completed, the agent reports these results to the simulator, which updates its environmental view accordingly to maintain a consistent state. For example, in section 5.2 we will see how agents fuse sensor data from disparate sources to produce a estimated target position. This position is needed to determine how and when other agents subsequently gather their data. A simulated fusion of this data would be inadequate, because MASS is unable to provide the necessary domain knowledge needed to perform this calculation. An agent, however, could do this, and then report to the simulator its estimated quality, cost and duration of the analysis process. Both parties are satisfied in this exchange - the agent will have the necessary data to base its reasoning upon, while the simulator is able to maintain a consistent view of the results of activities being performed. Using this mechanism, agents may be incrementally improved to meet real world requirements by adding real capabilities piecemeal, and using MASS to simulate the rest.

The other important class of event is the message event, which is used to model the network traffic which occurs between agents. Instead of communicating directly between themselves, when a message needs to be sent from one agent to another (or to the group), it is routed through the simulator. The event's lifetime in the simulation event queue represents the travel time the message would use if it were sent directly, so by controlling the duration of the event it is possible to model different network conditions. More interesting network behavior can be modeled by corrupting or dropping the contents of the message event. Like execution events, the message event may also may be influenced by other events in the system, so a large number of co-occurring message events might cause one another to be delayed or lost.

To prevent non-deterministic behavior and race conditions in our simulation environment, we utilize a kind of "controlled randomness" to order the realization of events within a given time pulse. When all of the agents have completed their pulse activity (e.g. they have successfully acknowledged the pulse message), the simulator can work with the accumulated events for that time slot. The simulator begins this process by generating a a unique number or hash key for each event in the time slot. It uses these keys to sort the events into an ordered list. It then deterministically shuffles this list before working through it, realizing each event in turn. This shuffling technique, coupled with control over the random function's initial seed, forces the events to be processed in the same order during subsequent runs without unfairly weighting a certain class of events (as would take place if we simply processed the sorted list). This makes our simulation com-

pletely deterministic, without sacrificing the unpredictable nature a real world environment would have. That's how we control the simultaneity problem.

5   Experiences

5.1   Intelligent Home project

The first project developed with MASS was the Intelligent Home project [18]. In this environment, we have populated a house with intelligent appliances, capable of working towards goals, interacting with their environment and reasoning about tradeoffs. The goal of this testbed was to develop a number of specific agents that negotiate over the environmental resources needed to perform their tasks, while respecting global deadlines on those tasks. The testbed was developed to explore different types of coordination protocols and compare them. The goal was to compare the performance of specialized coordination protocols (such as seen in [17]) against generic protocols (like Contract-Net[1] and GPGP[8]). We hoped to quantitatively evaluate how these techniques functioned in the environment, in terms of time to converge, the quality and stability of the resulting organization, and the time, processing and message costs.

JAF and TÆMS were used extensively, to develop the agents and model their goal achievement plans, respectively. MASS was used to build a "regular day in the house" - it simulates the tasks requested by the occupants, maintains the status of all environmental resources, simulates agent interactions with the house and resources, and manages sensors available to the agents. MASS allowed us to that events occurred at the same time in subsequent trials, and the only changes from one run to the next were due to changes in agent behavior. Such changes could be due to different reasoning activities by the agents, new protocols or varied task characteristics.

The Intelligent Home project includes 9 agents (dishwasher, dryer, washing machine, coffee maker, robots, heater, air conditioner, water-heater) and were running for 1440 simulated minutes (24 hours). Several simulations were run with different resource levels, to test if our *ad-hoc* protocols could scale up with the increasing number of resource conflicts. Space limitations prevent a a complete report of the project here, more complete results can be found in [17]. Instead, we will give a synopsis of a small scenario, which also makes use of diagnosis-based reorganization [13].

A dishwasher and waterheater exist in the house, related by the fact that the dishwasher uses the hot water the waterheater produces. Under normal circumstances, the dishwasher assumes sufficient water will be available for it to operate, since the waterheater will attempt to maintain a consistent level in the tank at all times. Because of this assumption, and the desire to reduce unnecessary network activity, the initial organization between the agents says that coordination is unnecessary between the two agents. In our scenario, we examine what happens when this assumption fails, perhaps because the owner decides to take a shower at a conflicting time (i.e. there might be a preexisting assumption

that showers only take place in the morning), or if the waterheater is put into "conservation mode" and thus only produces hot water when requested to do so. When this occurs, the dishwasher will no longer have sufficient resources to perform its task. Lacking adaptive capabilities, the dishwasher could repeatedly progress as normal but do a poor job of dishwashing, or do no washing at all because of insufficient amounts of hot water. We determined that using a diagnostics engine the dishwasher could, as a result of poor performance observed through internal sensors or user feedback, first determine that a required resource is missing, and then that the resource was not being coordinated over - the dishwasher did not explicitly communicate its water requirements to the waterheater. By itself, this would be sufficient to produce a preliminary diagnosis the dishwasher could act upon simply by making use of a resource coordination protocol. This diagnosis would then be used to change the organizational structure to indicate that explicit coordination should be performed over hot water usage. Later, after reviewing its modified assumptions, new experiences or interactions with the waterheater, it could also refine and validate this diagnosis, and perhaps update its assumptions to note that that there are certain times during the day or water requirement thresholds when coordination is recommended. The MASS simulator allowed us to explore and evaluate this new approach to adaptation without the need for a tremendous investment in knowledge engineering to create a realistic environment.

5.2 Distributed Sensor Network

A distributed sensor network (DSN) stresses a class of issues not addressed in the IHome project. We are presented in this research with a set of sensor platforms, arranged in an environment. The goal of the scenario is for the sensors to track one or more mobile targets that are moving through that environment. No one sensor has the ability to precisely determine the location of a target by itself, so the the sensors must be organized and coordinated in a manner that permits their measurements to be used for triangulation. In the abstract, this situation is analogous to a distributed resource allocation problem, where the sensors represent resources which must be allocated to particular tasks at particular times. Additional hurdles include a lack of reliable communication, the need to scale to hundreds or thousands of sensor platforms, and the ability to reason within a real time, fault prone environment. In this section we will show how JAF was migrated to new simulation and hardware environments.

Several technical challenges to our architectures are posed by this project. It operates in real-time, it must work in both a foreign simulation environment (called Radsim) and on an actual hardware implementation, it must function in a resource-constrained environment, and handle communication unreliability. We were provided with Radsim as a simulator, obviating the need for MASS[2]. Radsim is a multi-agent simulation environment operating in the DSN domain. One or more agents inhabit its environment, each attached to a sensor node.

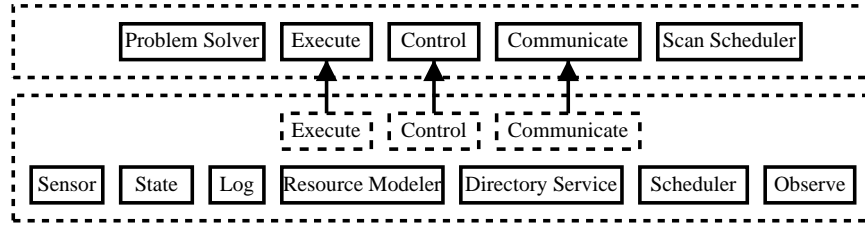[2] Radsim is developed and maintained by Rome Labs

41

Fig. 5. Organization of a DSN JAF agent. Upper bounds contain the domain dependent components, the lower bounds the independent.

Radsim models the communication between agents, the capabilities and action results of the individual sensors, and the position and direction of one or more mobile targets. Radsim differs from MASS in several significant ways. Because it is domain-specific, Radsim simulates a finite number of predefined actions, returning actual results rather than the abstract quality value returned by MASS. It's timeline is also continuous - it does not wait for pulse acknowledgement before proceeding to the next time slice. This, along with the lack of a standard seeding mechanism, causes the results from one run to the next to be non-deterministic. Our first challenge, then, was to determine what changes were required for JAF to interface with this new environment. This was done by deriving just three JAF components: Control, Communicate and Execute, as shown in figure 5. The new Control component determines the correct time (either in Radsim or hardware), the Communicate component funnels all message traffic through the radio-frequency medium provided in the environment, and additions to Execute provide the bridge allowing JAF actions to interface with the sensor. These changes were made with around 1,000 lines of code, the remainder of the JAF worked unchanged, allowing us to reuse roughly 20,000 lines of code. A domain dependent problem solver, which reasons about the various goals an agent should pursue, and scan scheduler, which produces scanning pattern schedules, were also implemented.
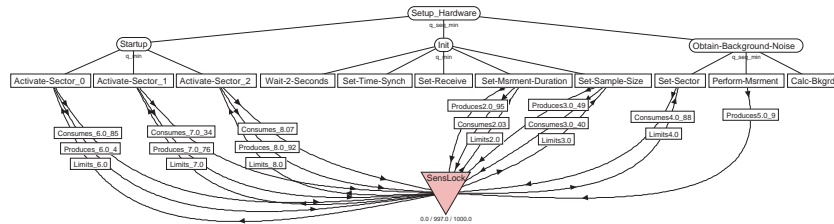

Fig. 6. TÆMS task structure for initializing a DSN agent.

To address real-time issues, the partial-order scheduler (mentioned in section 2) was used to provide quick and flexible scheduling techniques. A resource

modeling component was used to track both the availability of the local sensor, and the power usage of the sensor. This component was used by the scheduler to determine when resources were available, and to evaluate the probability that a given action might fail because of unexpected resource usage. The Communicate component was enhanced to add reliable messaging services (using sequence numbers, timeouts and retransmits), enabling other components to flag their messages as "reliable", if needed. Several new components were added to address the domain-specific tasks of scheduling the target detection scans, managing the track generation and performing the negotiation. In all cases there was a high degree of interaction between the new components and the generic domain-independent ones. Much of the necessary domain dependent knowledge was added with the use of TÆMS task structures, such as seen in figure 6. Here we see the initialization structure, which dictates how the agent should initialize its local sensor, perform background measurements, and contact its regional manager.

In our solution to this problem, a regional manager negotiates with individual sensors to obtain maximal area coverage for a series of fast target-detection scans. Once a target is found, a tracking manager negotiates with agents to perform the synchronized measurements needed for triangulation. Our technology enables this, by providing fine grained scheduling services, alternative plan selection and the capacity to remove or renegotiate over conflicting commitments. Two of the metrics used to evaluate our approach are the RMS error between the measured and actual target tracks, and the degree of synchronization achieved in the tracking tasks themselves.

After successfully demonstrating JAF in a new simulation environment, we were then challenged with the task of migrating it to the actual sensor hardware. In this case, JAF agents were hosted on PCs attached to small omnidirectional sensors via the serial and parallel ports. Our task was facilitated by the development of a middle layer, which abstracted the low level sensor actions into the same API used to interface with Radsim[3]. The actual environment, however, differed from Radsim in its unpredictable communication reliability, extreme measurement noise values and varied action durations. It also lacked the central clock definition needed to synchronize agent activities. In this case, the agents were modified to address the new problems, for instance by adding a reliable communication model to Communicate, and a time definition scheme to the control component. These JAF agents have been successfully tested in this new hardware environment, and we are currently in the process of developing better negotiation and scaling techniques to apply to this interesting domain [15].

5.3   Producer Consumer Transporter

The JAF/MASS architecture has also been used to prototype an environment for the to the producer, consumer, transporter (PCT) domain [13]. In this domain, there are conceptually three types of agents: producers, which generate resources;

---

[3] Middle layer API and sensor drivers were implemented by Sanders.

consumers, which use them; and transporters, which move resources from one place to another. In general, a producer and consumer may actually be different faces of a factory, which consumes some quantity of resources in order to produce others. There are several characteristics of this domain where alternatives exist for the factories and transporters - the choices made at these points by or for the PCT agents make up the organizational structure of the system.
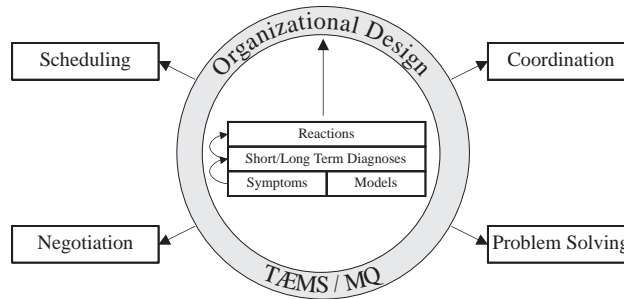

Fig. 7. Role of organizational knowledge within a PCT agent.

This particular system differs from the previous two in that it used the TÆMS representation as an organizational design layer, in addition to describing the local goals and capabilities of the agent. For instance, the subjective view would describe which agents in the system a consumer could obtain resources from, or identify the various pathways a transporter could take. It also made use of a third view of TÆMS - the *conditioned* view. The agent's conditioned TÆMS view is essentially the subjective view, modified to better address current runtime conditions. In figure 8, we see a subjective view which provides three potential candidates capable of producing X. Instead of specifying all producers a consumer could coordinate with, the conditioned view might identify only those which were most promising or cheap or fast, depending on the current goals of the agent. On the right side of the figure, we can see an example of this, where P1 and P3 have been removed from consideration. The idea is to constrain the search space presented by the task structure, to both speed up the reasoning and selection process, and increase the probability of success. The conditioned view was used as the organizational design for the agent - since the majority of decision making was based on this structure, changes in the organization could be made there to induce change in the agent's behavior.

In addition to local reasoning, a diagnosis component was used to generate the conditioned view. As mentioned earlier, the diagnosis component made use of a causal model, which served as a road map from symptom to potential diagnoses. The component itself would monitor the state of the agent, by listening to event streams and monitoring state, to detect aberrant behavior.
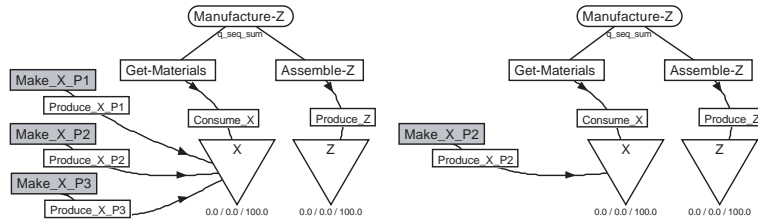
Fig. 8. Subjective and conditioned task structures for PCT

Once detected, the causal model would be used to identify potential causes of the problem, which would result in a set of candidate solutions. These solutions would be induced by making changes to the organizational design in the agent, through modification to the local conditioned view, as shown in figure 7. The JAF architecture facilitated this sort of technology, by providing the common mechanisms for interaction among components. The diagnosis component was integrated by simply plugging it into the agent, and no modifications to other components were necessary.
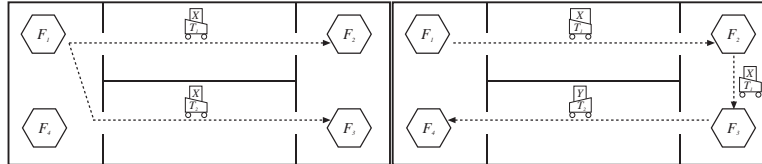

Fig. 9. Experimental solutions in the PCT environment.

Experiments in this environment focused on the convergence time of various diagnosis techniques to stable organizations, and the efficiency of those organizations. For instance, in figure 9, initial conditions in the environment on the left included transporters T_1 and T_2 bringing resource X from producer F_1 to consumers F_2 and F_3. Later in the scenario, the needs of F_4 change, such that it now requires Y. Several different organizations are possible, not all of them functional and efficient. Different diagnosis techniques were applied to situations like this to evaluate the characteristics of the individual organizations, and eventually converging on a solution like that shown on the right side of the figure. More details on the results of these experiments can be found in [13], and more sophisticated PCT environments are currently being tested and evaluated with the help of Dr. Abhijit Deshmukh and Tim Middelkoop.

6 Conclusions

The key idea in this article is the ability of the JAF/MASS architecture to quickly and easily prototype and explore different environments. Varied coordination, negotiation, problem solving and scheduling can all be implemented and tested, while retaining the ability to reuse code in future projects or migrate it to an actual implemented solution.

The JAF component-based agent framework provides the developer with a set of guidelines, conventions and existing components to facilitate the process of agent construction. We have seen in several examples how generic JAF components can be combined with relatively few domain specific ones to produce agents capable of complex reasoning and behaviors. The use of TÆMS as a problem solving language further extends the usefulness of this framework by providing a robust, quantitative view of as agent's capabilities and interactions. Of particular importance is JAF's demonstrated ability to easily work in a wide range of environments, including the discrete time MASS simulator, the real-time Radsim simulator, and on actual hardware, while making use of existing, generic components.

The MASS simulation environment was built to permit rapid modeling and testing of the adaptive behavior of agents with regard to coordination, detection, diagnosis and repair mechanisms functioning in a mercurial environment. The primary purpose of the simulator is to allow successive tests using the same working conditions, which enables us to use the final results as a reasonable basis for the comparison of competing adaptive techniques.

In the Intelligent Home project, we showed how a heterogenous group of agents were implemented in JAF and tested using MASS. Different coordination and problem solving techniques were evaluated, and the TÆMS language was used extensively to model the domain problem solving process. In the distributed sensor network project, JAF agents were deployed onto both a new simulation environment, and real hardware. Agents incorporated complex, partial-ordered scheduling techniques, and ran in real-time. Finally, in the producer/consumer/transporter domain, notions of organizational design and conditioning were added, and adapted over time by a diagnosis component.

We feel the main advantages of the JAF framework are its domain independence, flexibility, and extensibility. Our efforts in MASS to retain determinism without sacrificing unpredictability also make it well suited for algorithm generation and analysis.

7 Acknowledgements

implemented by Raphen Becker, Roger Mailler and Jiaying Shen, and Sanders and Rome Labs provided background, simulation and hardware expertise. Brett Benyo contributed to our work in the PCT domain.

References

1. Martin Andersson and Tumas Sandholm. Leveled commitment contracts with myopic and strategic agents. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 38–44, 1998.
2. K. S. Barber, A. Goel, and C. E. Martin. Dynamic adaptive autonomy in multi-agent systems. *Journal of Experimental and Theoretical Artificial Intelligence*, 2000. Accepted for publications. Special Issue on Autonomy Control Software.
3. Deepika Chauhan. *A Java-based Agent Framework for MultiAgent Systems Development and Implementation*. PhD thesis, ECECS Department, University of Cincinnati, 1997.
4. K. Decker. *Environment Centered Analysis and Design of Coordination Mechanisms*. PhD thesis, Department of Computer Science, University of Massachusetts, Amherst, 1995.
5. Keith Decker and Victor Lesser. Generalizing the partial global planning algorithm. *International Journal of Intelligent Cooperative Information Systems*, 1(2):319–346, 1992.
6. Keith Decker and Victor Lesser. Quantitative modeling of complex environments. Technical report, Computer Science Department, University of Massachusetts, 1993. Technical Report 93-21.
7. Keith S. Decker. Task environment centered simulation. In M. Prietula, K. Carley, and L. Gasser, editors, *Simulating Organizations: Computational Models of Institutions and Groups*. AAAI Press/MIT Press, 1996. Forthcoming.
8. Keith S. Decker and Victor R. Lesser. Generalizing the partial global planning algorithm. *International Journal of Intelligent and Cooperative Information Systems*, 1(2):319–346, June 1992.
9. Keith S. Decker and Victor R. Lesser. Quantitative modeling of complex environments. *International Journal of Intelligent Systems in Accounting, Finance, and Management*, 2(4):215–234, December 1993. Special issue on "Mathematical and Computational Models of Organizations: Models and Characteristics of Agent Behavior".
10. Keith S. Decker and Victor R. Lesser. Coordination assistance for mixed human and computational agent systems. In *Proceedings of Concurrent Engineering 95*, pages 337–348, McLean, VA, 1995. Concurrent Technologies Corp. Also available as UMASS CS TR-95-31.
11. Martha E. Pollack Hanks, Steven and Paul R. Cohen. Benchmarks, testbeds, controlled experimentation, and the design of agent architectures. *AI Magazine*, 14(4):pp. 17–42, 1993. Winter issue.
12. Bryan Horling. A Reusable Component Architecture for Agent Construction. UMASS Department of Computer Science Technical Report TR-1998-45, October 1998.
13. Bryan Horling, Brett Benyo, and Victor Lesser. Using Self-Diagnosis to Adapt Organizational Structures. Computer Science Technical Report TR-99-64, University of Massachusetts at Amherst, November 1999. [http://mas.cs.umass.edu/ bhorling/papers/99-64/].

14. Bryan Horling et al. The tæms white paper, 1999. http://mas.cs.umass.edu/research/taems/white/.

15. Bryan Horling, Régis Vincent, Roger Mailler, Jiaying Shen, Raphen Becker, Kyle Rawlins, and Victor Lesser. Distributed sensor network for real time tracking. Submitted to Autonomous Agents 2001, 2001.

16. Lyndon Lee Hyacinth Nwana, Divine Ndumu and Jaron Collis. Zeus: A tool-kit for building distributed multi-agent systems. *Applied Artifical Intelligence Journal*, 13(1):129–186, 1999.

17. Victor Lesser, Michael Atighetchi, Bryan Horling, Brett Benyo, Anita Raja, Regis Vincent, Thomas Wagner, Ping Xuan, and Shelley XQ. Zhang. A Multi-Agent System for Intelligent Environment Control. Computer Science Technical Report TR-98-XX, University of Massachusetts at Amherst, October 1998.

18. Victor Lesser, Michael Atighetchi, Bryan Horling, Brett Benyo, Anita Raja, Regis Vincent, Thomas Wagner, Ping Xuan, and Shelley XQ. Zhang. A Multi-Agent System for Intelligent Environment Control. In *Proceedings of the Third International Conference on Autonomous Agents*, Seattle,WA, USA, May 1999. ACM Press.

19. Nelson Minar, Roger Burkhart, Chris Langton, and Manor Askenazi. The swarm simulation system: A toolkit for building multi-agent simulations. Web paper: http://www.santefe.edu/projects/swarm/, Sante Fe Institute, 1996.

20. Nortel Networks. Fipa-os web page. Web, 2000. http://www.nortelnetworks.com/products/announcements/fipa/.

21. Régis Vincent, Bryan Horling, Victor Lesser, and Thomas Wagner. Implementing soft real-time agent control. Submitted to Autonomous Agents 2001, 2001.

22. Thomas Wagner, Alan Garvey, and Victor Lesser. Criteria-Directed Heuristic Task Scheduling. *International Journal of Approximate Reasoning, Special Issue on Scheduling*, 19(1-2):91–118, 1998. A version also available as UMASS CS TR-97-59.