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

* Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory Air Force Materiel Command, USAF, under agreement number #F30602-97-1-0249 and #F30602-99-2-0525. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. This material is based upon work supported by the National Science Foundation under Grant No.IIS-9812755. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Air Force Research Laboratory or the U.S. Government.

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

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.

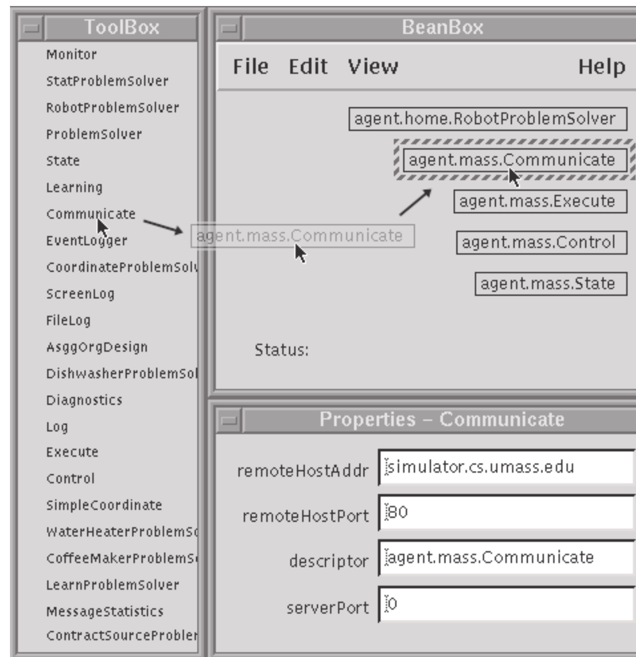

Fig. 1. Sun Beanbox, which can be used to build JAF agents.

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.

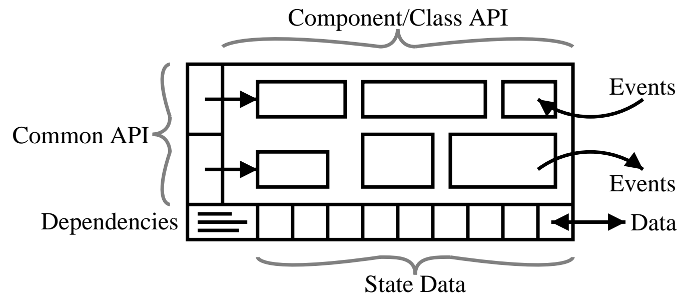

Fig. 2. Abstract view of a typical JAF component.

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 interaction 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 schedule, 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, monitoring 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 interactions. 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 scaffolding, 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 different 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 possible 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.

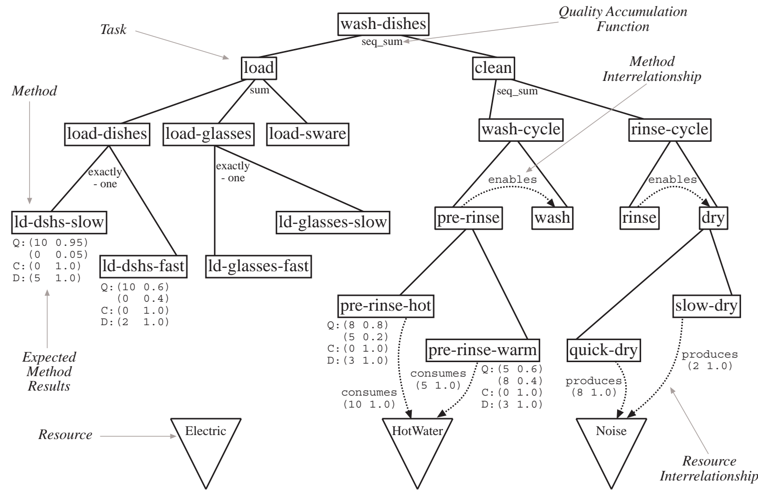

Fig. 3. TÆMS task structure for the IHome Dishwasher agent

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.

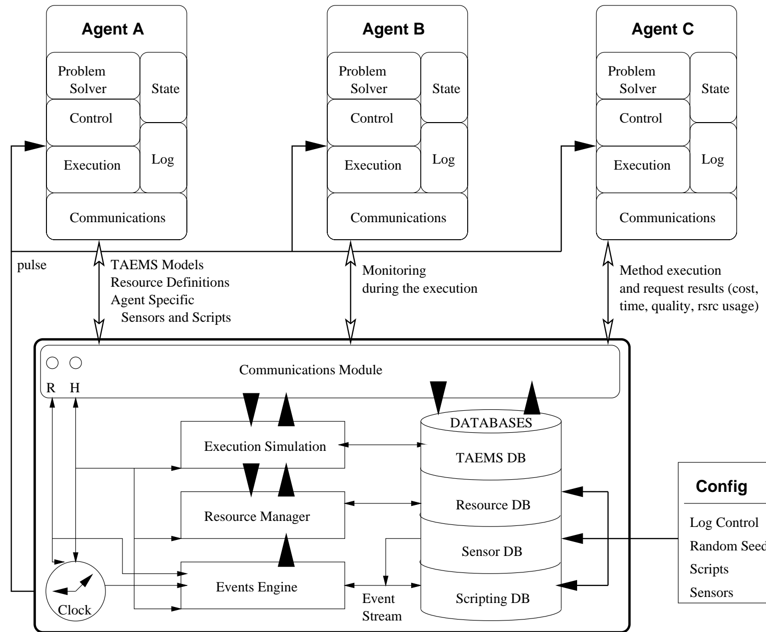

Fig. 4. Architecture of the MASS and agent systems.

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

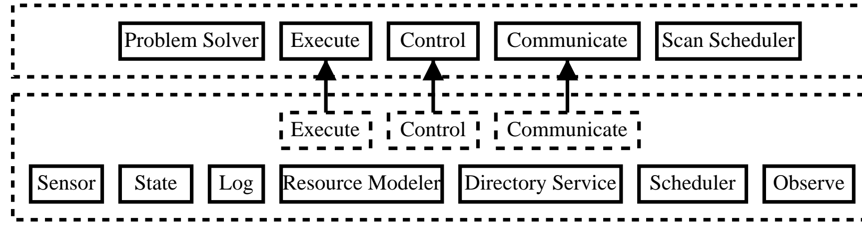

Fig. 5. Organization of a DSN JAF agent. Upper bounds contain the domain dependent components, the lower bounds the independent.

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.

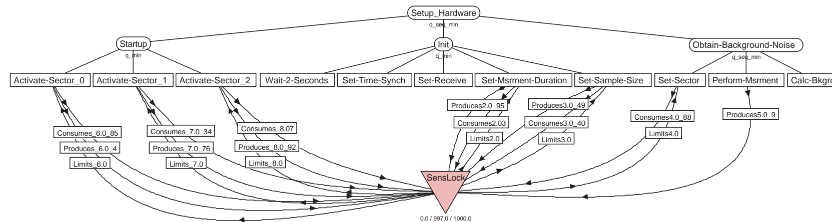

Fig. 6. TÆMS task structure for initializing a DSN agent.

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.

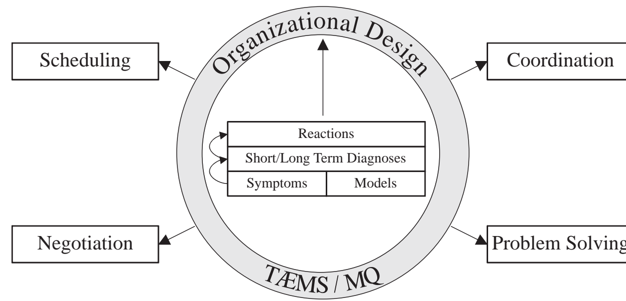

Fig. 7. Role of organizational knowledge within a PCT agent.

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.

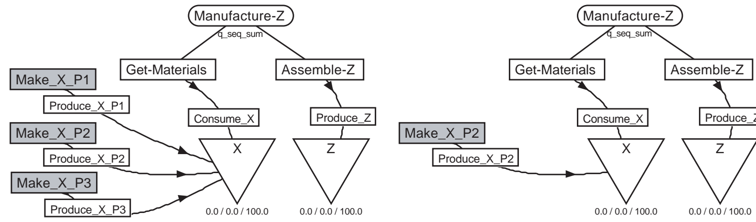

Fig. 8. Subjective and conditioned task structures for PCT

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.

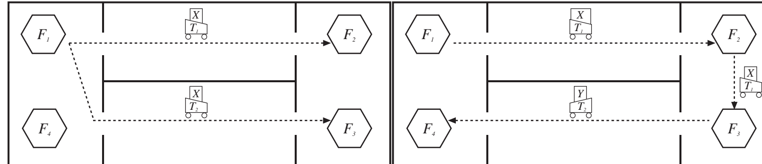

Fig. 9. Experimental solutions in the PCT environment.

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

We wish to thank the following individuals for their additional contributions to the research described in this paper. Thomas Wagner contributed the Design-To-Criteria scheduler and to extensions to the TÆMS formalization. Michael Atighetchi, Brett Benyo, Anita Raja, Thomas Wagner, Ping Xuan and Shelley XQ. Zhang contributed to the Intelligent Home project. The DSN project was

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.