Experiences in Simulating Multi-Agent Systems Using TÆMS*

Régis Vincent, Bryan Horling, Victor Lesser
Department of Computer Science
University of Massachusetts, Amherst, MA, 01003
Email: vincent,bhorling,lesser@cs.umass.edu, Phone: +1 413 545 0675

Abstract

Building multi-agent systems in real, open environments, often means we have to spend significant energy on resolving issues orthogonal to the initial goals of the research. To avoid such overhead many researchers choose to implement, test and evaluate their multi-agent systems in a simulated world. In addition to providing a better-defined and predictable debugging environment, a good simulator can also help evaluate and quantify aspects of multi-agent system and multi-agent coordination in a controlled environment through repeated experiments. In this paper we will describe the simulator we built, which uses the TÆMS modeling language as its primary knowledge representation. We will also present two multi-agent environments built using these tools and how the simulator can be used to effectively implement and test multi-agent systems.

1 Motivations for a Simulated World

As researchers in multi-agent systems, we hope to build, deploy, and most importantly evaluate multi-agent systems in real, open environments. Unfortunately,

* Effort partially 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. 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. Disclaimer: 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, National Science Foundation, or the U.S. Government.

working in such environments usually implies we expend significant energy on resolving issues orthogonal to the initial goals of the research, such as dealing with the knowledge engineering and low-level system integration issues. To avoid such overhead many researchers choose to implement, test and evaluate their multi-agent systems in a simulated world. In addition to providing a better-defined and predictable debugging environment, a good simulator can also help evaluate and quantify aspects of multi-agent system and multi-agent coordination in a controlled environment through repeated experiments.

To be useful, a simulation environment should be carefully designed Hanks et al. define in [9] several characteristics a multi-agent simulator should possess. It should permit *exogenous* or unplanned events to occur. Because the system is closed, environmental *complexity* is required to make the simulation more realistic and applicable; ideally, the simulated world should react based on prior observations of the real world. For example, a network's behavior may be based on actual network statistics. *Quality and cost* of sensing and effecting need to be explicitly represented in the testbed to simulate imperfect sensors and activators. It should also have a clean and reasonably complete interface that allows agents to "observe" the world. *Multiple agents* must be able to act and interact in the simulated world. A *clean interface* between the agents and simulation controller is necessary. We elaborate on this by claiming that agents and simulator should run in separate processes or even different processors when possible - the communication between agents and simulator should not make any assumptions of the simulation's configuration, such shared file systems or memory. A well-defined model of *time* is needed to make simulation deterministic. Events are defined by their occurrence in simulation time, which may be independent of real time.

We add to this list two other characteristics: *deterministic experimentation* and different *worldviews*. *Deterministic experimentation* is needed to compare different trials, as one must be sure what parameters vary between them. By allowing agents to have different *worldviews*, the designer can directly explore the ramifications of agents possessing incorrect beliefs. The second involves the ability to introduce uncertainty into agents' beliefs what may come from either incorrect or approximate or incomplete knowledge. The manipulation of agents' uncertainty we feel is the key to study how a MAS will function in an open, real environment.

Most of the existing MAS simulators concentrate on a state-based simulation, they are responsoble for simulating the world and the effects of the agents' actions. Simulators like SWARM ([12]) or MANTA ([5, 6]) were designed to simulate MAS with independent agent's actions. MAGSY ([7]) and PHOENIX [10]) on the other hand were designed to simulate interacting actions, but agent coordination is completely independent of the simulation. Because, we want to explicitly study agent coordination and evaluate protocols in different classes of problems,

none of those simulators fit our needs. We have used past experience [2, 13] to build a better environment.

In this paper we will describe the simulator we built, which uses the TÆMS modeling language as its primary knowledge representation. We will also present two multi-agent environments built using these tools and how the simulator can be used to effectively implement and test multi-agent systems.

2 TÆMS as a Simulator Knowledge Representation

TÆMS[4] is a task modeling language that can be used to represent agent activities. It models planned actions, candidate activities, and alternative solution paths from a quantified perspective, by using a task decomposition tree. In this design, the root nodes of the structure, or task groups, represent goals the agent can achieve. Internal nodes, or tasks, represent sub-goals and provide the organizational structure for primitive executable methods, which reside at the leaves of the tree. Each method is characterized along three dimensions: quality, which the agent hopes to maximize, cost, which the agent tries to minimize, and duration, which describes the time required to executing the method. The dimensions themselves are discrete distributions, so each probability/value pair represents a potential result for the characteristic in question. In structure TÆMS is very close to HTN (Hierarchical Task Network) or TCA (Task Control Architecture) approaches. TÆMS extends the HTN approach by replacing the logical decomposition (AND and OR) of tasks by a continuous function (called *quality accumulation function* or *qaf*) that defines how to accumulate the quality dimension based on the sub-task quality. For example, instead of having AND decomposition that require all the sub-tasks to be completed, TÆMS defines the similar decomposition *q_min* that also require all the sub-tasks to be completed but also specify that the quality of the task is the minimum of all sub-tasks quality. Another important ability of TÆMS is its capacity to represent relationships between methods or tasks. For example, if method A must successfully complete before B can be started, we say that A *enables* B. Other interrelationship types include *disables*, *hinders*, and *facilitates*. Any of those relationships could be defined locally in an agent or across agents. Resources consumed or produced by tasks and methods can be represented using a different kind of interrelationship, which quantifies those effects. The effects that can occur by exceeding the bounds on the use a resource may be similarly described. In addition to describe characteristics local to the agent, TÆMS models also permit the designer (or agent) to represent elements associated with other agents. Thus, this same structure can allow the agent to reason both about its own internal state and as well the potential capabilities and interactions of other agents.

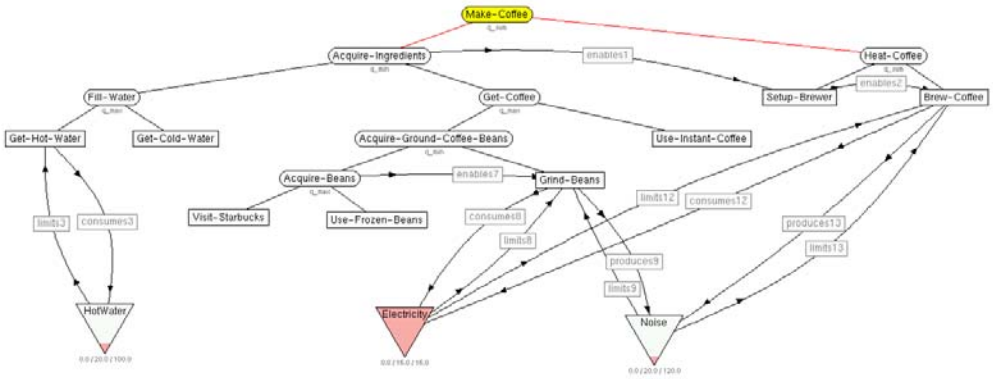

Figure 1: TÆMS Task Structure for Making Coffee

Figure 1 shows an example of a TÆMS task structure, describing the abilities of a coffee maker agent in a simulated intelligent home populated with agents controlling different appliances and resources [11]. According to this model, the agent has a single goal of **Make-Coffee**, which can be achieved by acquiring the ingredients and then heating the coffee. The *qaf* for Make-Coffee is *q_sum*, which indicates that the final quality of this task is the sum of **Acquire-Ingredients** and **Heat-Hoffee**. On the left side of the figure one can see that as part of acquiring ingredients, the agent must obtain water, which it can do in two different ways: **Get-Hot-Water** and **Get-Cold-Water**. The *qaf* associated with **Fill-Water** is a *q_max* which take the maximum of the two sub-tasks (*q_min* is equivalent to an *OR*).These methods represent primitive actions the agent can perform, and are described in terms of their quality, cost and duration distributions. Using these descriptions, a scheduling or control component could reason about the tradeoffs of each action, and choose the most appropriate one given the agent's current context. Once the agent has obtained water, it should also get the coffee. We have defined two ways to acquire coffee beans, from Starbucks and by using frozen beans. The task of acquiring coffee beans *enables* the fact that the agent may grind them, i.e. you can't grind the beans until the agent has acquired them. Of course, if the agent is in a hurry, it can also just use instant coffee. The final step is to brew the coffee itself.

TÆMS can be used both as a subjective internal representation for agent reasoning or as an objective modeling language for formally specifying a problem. At the same time, we have found that it may also serve as a compact world representation for a simulation environment. An agent in such an environment will have several versions of the same TÆMS structure, which it uses internally. The

simulation controller would have an equivalent task structure, which it will use to determine and enforce the correct world model. The fact that the agents and simulator's view can be different also allows us to model situations where agents reason about potentially incomplete or incorrect descriptions of the world. In these cases, we say the agent has a *subjective* view it perceives to be true, while the simulator uses an *objective* view that is the actual truth. This is how we model and simulate uncertainty in the agent belief and we do believe that it's a key feature when designing systems for working in dynamic, unpredictable environments.

TÆMS was developed originally to support agent reasoning, and to describe instances where agents might need to coordinate over interrelated tasks. We also found that it was a very compact representation for multi-agent simulation environments because the relationships between objects (task, method, and resources) were explicit. So if a particular action an agent might perform would consume five units of resource A, it would have an explicit consumes interrelationship between that method and resource A. In the next section, we will describe the TÆMS based multi-agent system simulator we have built.

3 Multi-agent system simulator (MASS)

The Multi-Agent System Simulation MASS is a more advanced incarnation of the TÆMS simulator created by Decker and Lesser in 1993. MASS 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 a domain-independent, hierarchical representation of an agent's goals and capabilities (see Figure 2), to represent their knowledge. To prevent any interference or assumption, MASS and the agents are running in separate different processes, which provides a more accurate model of how they would function under real conditions. This TÆMS model is also used to supply the internal domain knowledge of the simulation controller. This fact, coupled with a simple configuration mechanism and robust logging tools, make MASS a good platform for rapid prototyping and evaluation of multi-agent systems. From an agent's point of view, MASS is acting like the world, we have made the simulation completely transparent for the agent. We have derived some basic agent's components (like the communication module or the execution module) to interface with the simulator but no other components in the agent are aware of the simulator. Using this object-oriented technique of subclassing offer a direct and immediate transition from a simulated agent to a real agent by just replacing the MASS-aware component by a normal one.

Figure 2 shows the overall design of MASS, and, at a high level, how it in-

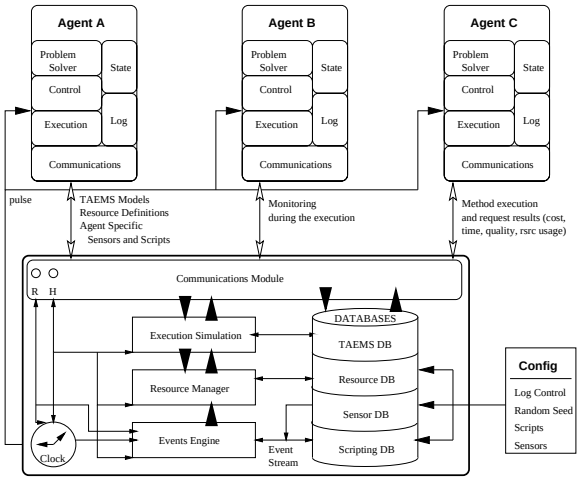

Figure 2: Architecture of the MASS system.

teracts with the agents connected to it. On initialization, MASS reads 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. 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. 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 the figure. By doing the assimilation, the simulator constructs the *global objective* TÆMS task structure. This *global objective* view will be used by the execution subsystem when the simulation is running to quantify both the characteristics of method execution and the effects interactions with resources and other actions 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 (see section 3.1.2). 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 the agents and destinations within the controller. In the database part of the figure, we will describe in details the *external event* generation (section 3.2) and the *sensing* mechanism (section 3.3).

The MASS controller has several tasks to perform while managing simulation. These include routing message traffic to the correct destination, providing hooks that allow agents to sense the virtual environment and managing the different resources utilized by the agents. Its primary role, however, is to simulate the exe-

cution of methods requested by the agents. The simulator uses the distributions in its global objective view to compute the values for method execution in terms of cost, quality, duration and resource consumption. Note also that those values are the typical case outcomes, the simulator will degrade results as necessary (lower quality, more cost or longer duration) if required resources are not available, or if execution is adversely affected by interactions with other methods. For example, the method **Visit-Starbucks** has the following distribution: 50% of the time it has a duration of 15 minutes and 50% 20 minutes[1]. In a more physical sense, this means that sometimes you can get to Starbucks in 15 minutes, and sometime you need 20 minutes. We don't want to model road traffic, weather, or other possible disturbances in this scenario, the action is simplified to its base components. Based on this distribution, MASS will pick a value (randomly choosen based on the frequency, or density, of each possible result) - let's say 15 minutes in this case. The simulator will then simulate the execution of **Visit-Starbucks**, which will normally complete in 15 minutes. It is important to note that at any time during the execution of the method, it is possible for an external event to affect that duration (or quality or cost). For example, it may be the case that another agent in the environment has performed an action that will hinder our trip to Starbucks in some way, adding two minutes to the total execution duration. The agent who had requested this execution will now be notified of its completion 17 minutes after starting it, instead of 15. The choice of how to handle this particular problem is left up to the agent itself.

The second responsibility consuming a large portion of the simulator's attention is to act as a message router for the agents. Agents send and receive their messages through the simulator, which allows us 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 3.1.2 will describe this mechanism in more detail.

3.1 Controllable Simulation

In our simulated experiments, our goal is to compare the behavior of different algorithms in the same environment under the same conditions. To correctly replicate running conditions for a controlled simulation, the simulation should have a deterministic notion of time, a deterministic notion of randomness and a deterministic order of events. Two simulation techniques exist which we have exploited to achieve this behavior: discrete time and events. Discrete time simulation segments

[1]The duration in TÆMS is actually a unitless value, we represent times in minutes here for clarity.

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 a period of (real) CPU time. In our model, a pulse does not represent a predefined quantity CPU time; each agent decides independently when to stop running; this allows agent performance to remain independent of the hardware it runs on. The second type of simulation is event-based, which means that the control is directed by events that force agents to react. The MASS simulator combines these 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.

3.1.1 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.

3.1.2 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). 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 by the agents in our environment.

All 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 2). The execution event will also calculate the other qualities associated with a method's execution, such as its cost, quality and resource usage. All interrelationships that could affect this method (for example *enables* from another agent) are checked and their effects integrated with the current method distribution. 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.

Real activities may be also incorporated into the MASS environment by allowing agents to notify the controller when it has performed some activity. Using this mechanism, an agent may execute some method locally, generating actual results in the process. When completed, it then reports these results to the simulator, which updates its local view accordingly to maintain a consistent state. Agents may then be incrementally generated 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 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 completely deterministic, without sacrificing the unpredictable nature a real world environment would have.

3.2 External Event Scripting

MASS includes an event scripting mechanism that permits the creation of a wide range of events capable of changing simulation characteristics. Using this mechanism, event generation can be triggered by any combination of characteristics such as time, resource's state, agent activity or sensor data. Script effects may generate

arbitrary network data, disconnect agents or even modify their internal state. The reactions we have found to be most useful are resource related. Using a script, it is possible to change the state of any resource for an arbitrary period of time, including its current level and upper and lower bounds. Scripting also provides an additional mechanism by which we can simulate both external influences and internal agent behaviors. The following scenarios demonstrate how this can be accomplished:

- Change the maximum/minimum usage bound for a resource for some period of time. This can be used to simulate a resource that has temporal capacity fluctuations, as is seen in network bandwidth.
- Permanently alter the maximum/minimum usage bound, or currently available quantity, to model an upgraded or damaged resource.
- Continuously adjusts a resource's level value over a period of time, or until some other event occurs. This can be used to simulate a leaky resource, or one being used by a non-agent entity, and the results of a repair mechanism.
- Change resource sensor characteristics, simulating damage or repair to that component.
- Modify or insert a new TÆMS task structure in an agent, which might indicate a new set of goals or a more refined view of local, remote or environmental capabilities. This is particularly useful for simulating planner, plan retriever or domain problem solving capabilities that are not implemented in the agent. The designer of the simulation can generate appropriate structures a-priori which are then supplied to the agent by the simulator. For the agent, those new structures are treated as if they were generated locally at runtime.
- Modify the execution characteristics of a particular method, to mimic the effects of an adversarial intrusion.

3.3 Sensor Simulation

To provide sensing capabilities, we have defined a mechanism that allows agents to read environmental information from the simulated world via the controller. Our goal was to make the interface generic, so agents could actively gather information on subjects ranging from the current time to the current noise level in a room to the location of another agent in the system. Within the simulator one or more sensor objects can be created, each of which can access a particular environmental

characteristic with some amount of precision. Sensors that return arbitrary data (as number of agents in the simulation, address of a matchmaker) are also possible.

Multiple sensor objects may be defined to access the same environmental characteristic, but with different precision levels and agent access privileges. This allows us to model different sensing capabilities and damaged or faulty sensors, and explore scenarios involving redundant and/or conflicting sensor results.

As an example, suppose the Dishwasher agent in the intelligent Home is able to sense the current noise level, but the simulator will return the sensed level adjusted by up to 5% of its real value. Meanwhile, the Vacuum agent might read the value with perfect precision. So if the current level is 76 dB, the Dishwasher can receive a reading anywhere from 72.2 to 79.8, while Vacuum will always know the correct value of 76.

4 Experiences

Using TÆMS, we have developed several simulation environments, including an Intelligent Home system, and a new (still under development) supply chain system involving 20 agents. In this section, we will show how we used MASS and TÆMS for those two testbeds.

4.1 Intelligent Home project

The first project developed with MASS was the Intelligent Home project. 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 specific idea was to compare the performance of specialized protocols (such as seen in [11]) against generic protocols (like Contract-Net[1] and GPGP[3]). 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. Using MASS features, we can guarantee that events always occur at the same time in subsequent trials, and the only changes from one run to the next are due to changes in agent behavior. Such changes could be due to new reasoning activities by the agents, new protocols or new task characteristics. The designer, however, must explicitly make all those changes, so their impact may be easily quantified and

measured.

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 [11]. Instead, we will give a synopsis of a small scenario, which also makes use of diagnosis-based reorganization [**?**].

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 adaptative capabilitities, 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.

4.2 Supply Chain

The supply chain is a new project that involves up to 20 agents. Each agent represents a factory, which produces resources by consuming other resources. The problem is thus one of obtaining the necessary components from other factories in the environment, which in turn may need to obtain other resources for their local production. Each factory bids on contracts for supplying other factories, potentially cascading their needs to others factories. The simulator, by generating requests for final products supplies the ”consumer” demand. In our testbed, we have used the PC manufacturing industry. A Consumer will order a number of PC systems per day, through distribution center like *Best Buy* or *CompUSA*. If those distribution centers don't have sufficient local reserves to satisfy the request, they will react by sending other bids for other resources required satisfying the requests.

In this testbed, MASS allows us to simulate different agents at different locations, which forces agents to take account transportation between locations. MASS natively defines the notion of a locale (a spatial location, potentially associated with local resources and sensors), so each agent will be in a different locale with its own storage for goods. Locales in MASS can also be hierarchical; sub-locales can be defined to specialize where resources are stored. This could be as complex as needed, but our initial approach to the supply chain testbed is to break down locals by town, in which factories may be located. The connections allowed between locales are arbitrary; in this case, we have defined two types of connections: *highways* and *airlines*. The type of connection that is used to ship goods will affect the shipment price and duration. MASS is able to compute the expected cost, duration and quality for transferring goods, depending of the connection you will use, and external events may also affects those values (to simulate a snow storm at an airport, for instance).

We envision this testbed growing up to 100 agents, which will MASS to be extended to support such large communities. To address this issue, we plan on distributing MASS as a hierarchy or graph of simulators, synchronized from one primary controller.

5 Conclusions and extensions

MASS was successfully used to develop and evaluate multi-agent system up to 20 agents. Even if those simulations work, they are involving a limited number of agents. Our next goal is to build a system with more than 100 agents using a distributed version of MASS. We are also working on integrating MASS and the DECAF [8] architecture from the University of Delaware to allow people to use MASS with their own agent architecture.

Based on those experiments, we found that MASS has a lot of necessary features for multi-agent systems: using the scripting language built-in has allowed us to have only single point of configuration for the simulation. It was also easy to use the scripting language to simulate component we didn't have like a domain problem solver (we use the simulator to send to an agent some new TÆMS task structure based on its previous execution). It was also useful for debugging to be able to rerun the same simulation over and over.

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. Decker, M. Williamson, and K. Sycara. Modeling information agents: Advertisements, organizational roles, and dynamic behavior. In *Proceedings of the AAAI-96 Workshop on Agent Modeling*, 1996.

[3] 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.

[4] 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.

[5] A. Drogoul and J. Ferber. Multi-agent simulation as a tool for modeling societies: Application to social differentiation in ant colonies. In *Europeen Workshop Modelling Autonomous Agents Multi Agent World (MAAMAW)*, pages pp. 3–23, Rome, Italy, 1992.

[6] Jacques Ferber. *Foundations of Distributed Artificial Intelligence*, chapter Reactive Distributed Artificial Intelligence: Principles and Applications, pages pp. 287–314. John Wiley & Sons, Inc., 1996.

[7] K. Fisher, J. Muller, and Pischel M. *Foundations of Distributed Artificial Intelligence*, chapter AGenDA - A General Testbed for Distributed Artificial Intelligence Applications, pages pp. 401–428. John Wiley & Sons, Inc., 1996.

[8] John R. Graham and Keith S. Decker. Towards a distributed, environment-centered agent framework. In *Proceedings of the Sixth International Workshop on Agent Theories, Architectures, and Languages (ATAL)*, Florida, jul 1999.

[9] 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.

[10] A. E. Howe. Analyzing failure recovery to improve planner design. In *Proc. of AAAI-92*, pages 387–392, San-Jose, CA, 1992.

[11] 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.

[12] 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.

[13] M. V. Nagendra Prasad and Victor R. Lesser. The use of meta-level information in learning situation_specific coordination. In *IJCAI-97*, Nagoya (Japan), 1997.