Generating problem episodes for a multi-agent system: The TAEMS Grammar Generator UMASS Computer Science Technical Report 1999-31

Brett R. Benyo, M.V. Nagendra Prasad, Victor R. Lesser

June 17, 1999

1 Abstract

Empirical studies are an important method of determining the roles of different algorithms in multi-agent systems research. In order to run multi-agent experiments, one needs a method of generating problem instances for the agents to solve. These problem instances can then be converted into TAEMS task structures, which are hierarchical abstractions of the problem solving process that describe alternative ways of accomplishing the desired goal. Most real application domains constrain the morphology of the task structures, therefore in order to generate appropriate problem instances, a model of the environment that the multi-agent system is operating in is needed. We present the TAEMS grammar generator, a environment modeling tool capable of capturing the constraints present in the domain task structures, along with the uncertain and dynamic nature of most real world environments.

2 Introduction

Multi-agent problem solving is becoming a new computing paradigm and presents many new challenges. A multi-agent system is a collection of entities called agents that use their knowledge and beliefs about their environment to perform actions. In this work, we are primarily interested in cooperative groups of semi-autonomous agents that coordinate with each other to achieve a global goal. Knowledge engineering is an important part of constructing such a system, for we have to specify what knowledge each agent has, what each agent can do, and endow the agents with knowledge and capabilities to respond to dynamic environments.

In order to represent the knowledge an agent has and the actions an agent is capable of, we use the TAEMS (Decker & Lesser 1993) modeling language. Using TAEMS, we can specify not only the actions an agent has available, but also how these actions affect each other and the environment. In addition, we can represent other agents and their possible actions. Our representation of actions gives the agents a statistical view of the possible outcomes of the actions, and an idea of how these outcomes can vary over time. One limitation of a TAEMS representation is that TAEMS does not have strong support for representing contingency. A TAEMS task structure is more of a static snapshot of the problem an agent faces at a given time, it is not a representation of all the things that could happen. To get around this limitation, we can use the generator to represent the dynamic nature of real world environments.

Problem solving begins when an agent generates or receives a goal tree called a TAEMS task structure together with performance objectives that could include hard deadlines. The top level of this tree represents the main goal that the agents need to achieve. Successive levels represent subgoals that can be combined in some way to achieve the higher level goal. The bottom most level (leaves of the tree) represent individual actions that the agents can perform. There may be many different subsets of actions that an agent can perform to achieve a top level goal, each with different quality, cost, and durations. It is the agent's job to determine which set of actions to perform to best solve the top level goal given the stated performance objectives. This determination process usually involves coordination among the agents. Figure 2 is an example of a TAEMS task structure, and a detailed description of TAEMS can be found in section 3.

A set of TAEMS task structures that represent the actions of all agents in the system is called an *episode*. Each task group in the episode has an arrival time, and possibly a deadline. No quality can be accumulated before the arrival time, or after the deadline. The role of the generator is to create an episode from a model of the environment. The environment is modeled as a generator function that takes a set of parameters and uses graph grammar productions to create episodes.

We present a graph grammar based task structure generator powerful enough to model the topological relationships occurring in task structures representing many real life applications. The generator can operate in three different modes. First, the generator can generate tasks from a centralized perspective. In this case, the generator is in a sense simulating the environment by giving tasks to the agents to perform. The generator can also generate tasks from an agent's perspective. In this case, the generator is acting as a component of an agent. The task structure it creates then represents the tasks that the agent has decided to undergo on its own based on, for instance, task recognition, sensor events, or requests for goals to be performed by other agents. Finally, the generator can operate in a dynamic mode, inserting events into the system that can cause

existing task structures to be modified. These event rules of the generator can be used to simulate a dynamic component of a domain problem solver, such as a learning component. The generator allows a researcher to quickly run experiments in different domains without the need to create a domain specific problem solver and environment model.

This generator is implemented in Java, and can be run as a stand-alone component, or in conjunction with our MASS multi-agent survivability simulator (Vincent, Horling, Wagner & Lesser 1997). The simulator is domain independent, and thus is capable of simulating any environment that can be represented in the TAEMS language. We have used it to simulate many environments such as an intelligent home (Lesser et al. 1998), a robotic delivery system, and an Internet based portfolio management system. The generator operates as an agent component that communicates with the simulator, providing the simulator with a statistical description of the outcomes of each possible agent action. This statistical description called the objective view, or the objective task structure. The generator also provides each agent with what is called a subjective view or subjective task structure. The subjective view may not be completely accurate, since it represents the agent's beliefs about the tasks. For example, an agent may believe that a certain action will be completed in five units of time, when in fact the action will take ten units.

Figure 1 shows the system architecture of the simulator. "The generator" is actually a set of generator components, each fully functional. Each generator component that is responsible for simulating the affect of the environment on its agent by providing that agent with tasks to perform. Alternatively, one generator component can be singled out to simulate the whole environment, and provide tasks to all other agents. These generator components that are also responsible for informing the simulator of tasks that the agent has decided to perform.

Work on the generator started in 1996 (Prasad, Decker, Garvey & Lesser 1996) in response to a desire to continue the work by Decker (Decker & Lesser 1995) on studying the roles of different coordination mechanisms in different task environments. The original generator created in 1996 (Prasad 1996) was used to produce static task structures. After creating the task structures, the generator would shut down. In order to study survivable systems, we needed a generator that is capable of modeling a more dynamic environment. Thus, work since 1996 has been focused on the dynamic capabilities of the generator, and integrating the generator with the MASS environment. Through the use of event rules and triggers, the generator can modify task structures while they are being executed by agents by sending events to the simulator. In addition, the generator can add, omit, or change portions of a task structure before delivering it to an agent to simulate the agent having incomplete or incorrect knowledge of the environment. These capabilities also allow the generator to be used in a completely different context; the generator be used to simulate an agents do-

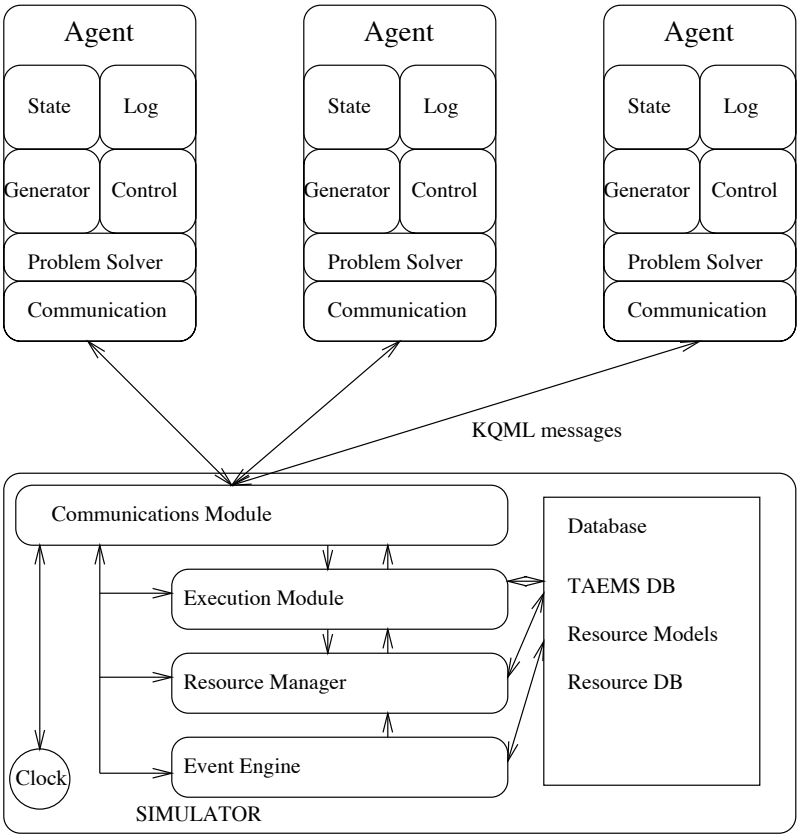


Figure 1: System architecture of the MASS simulator

main specific problem solving component or learning component.

In sections 3 and 4, we will briefly introduce TAEMS and graph grammars. In the fourth section, we will discuss the enhancements that were made to the generator to support dynamic and uncertain environments. These sections are intended to make the reader familiar with the concepts, and not necessarily get bogged down by the implementation details and the generator syntax. In Section 6, we give a description of the input and output of the generator, with the detailed syntax. Specific examples are given the following section.

3 TAEMS

TAEMS (Task Analysis, Environment Modeling, and Simulation) (Decker & Lesser 1995; Decker 1995) is a formal domain-independent language for representing tasks and problems. A TAEMS model of an environment specifies the actions that are available to the agents and relationships between those actions. In addition, the performance of all actions are described through discrete probability distributions in terms of quality, cost, and duration. A TAEMS task structure is a graph (usually a tree, however it is not required to be a tree) with a single root node, called the *task group*, or the *top level task*. Internal nodes in the graph are TAEMS *tasks*, and leaf nodes are executable TAEMS *methods*. The methods represent actions that can be executed directly by the agents. Each method is described by its quality, cost, and duration distributions. When a method is executed, its quality, cost, and duration are chosen by the simulator from these distributions. The results of methods can then be combined into the more abstract tasks, through a quality accumulation function.

Figure 2 gives an example of a TAEMS task structure representing the activities of a Internet information gathering agent. The top level task, Process-Query represents the problem at the most abstract level. The problem presented to this agent is that of responding to a request for information from an outside source, such as a user-interface agent. Completing the Process-Query task involves completing the three subtasks, Formulate-Plan, Get Data, and Return-Results. Formulate-Plan and Return-Results are terminal methods, which means that they are represented at their most detailed level, and thus can be executed directly by the agent. The effect that executing these method has is represented by a TAEMS outcome, depicted by the quality, duration, and cost distributions shown underneath the method box. The Formulate-Plan method has a 100 percent chance of returning a quality of ten, taking five units of time to complete, and incurring no monetary cost. The quality accumulation function of a task tells how to combine the quality of its subtasks to derive the quality of the parent task. The quality accumulation functions in Figure 2 are printed directly below the task boxes. For Process-Query, the function is "q-min", meaning that the quality of Process-Query is the minimum of the

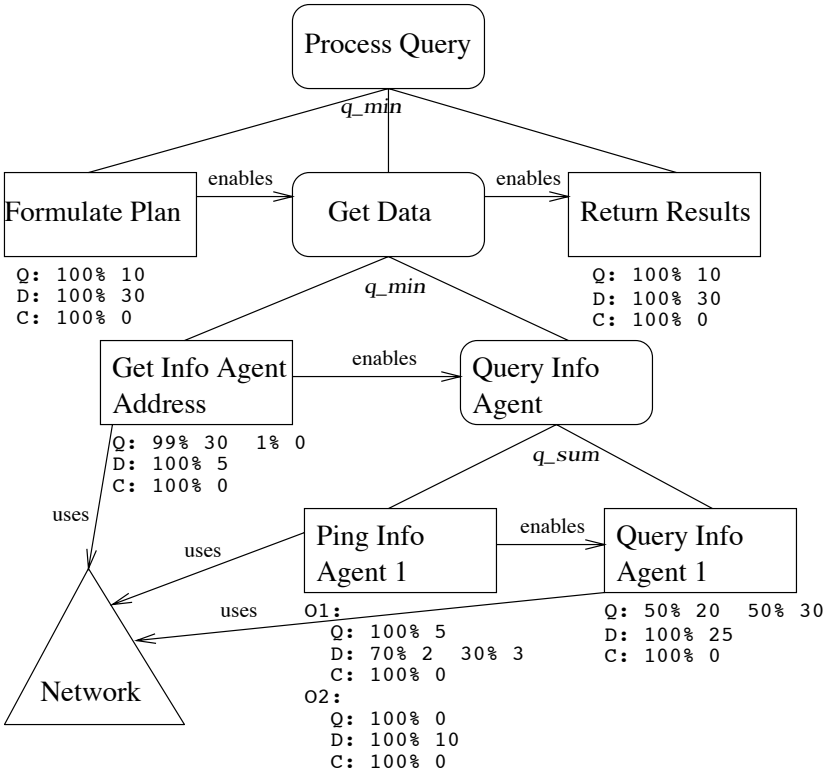


Figure 2: TAEMS task structure for a information gathering agent

qualities of its three subtasks. This means that all three subtasks must be executed for Process-Query to have any degree of success at all.

Get-Info-Address is a method that represents asking a matchmaking service for the address of an information source agent. The Get-Info-Address method has a one percent chance of returning a zero quality. Zero quality represents failure; the method has no degree of success at all. If this is the case, due to the "q-min" accumulation functions, there is no way that Process-Query can achieve any quality greater than zero. This represents the case that our agent is unable to get the address of any information source agents, and thus is unable to process the query. This is an example of the static nature of a TAEMS task structure. We have no direct way of saying in TAEMS, "If Get-Info-Address returns 0 quality, abort Process-Query and do something else" If we had another way of finding out the address of an information agent (such as asking a peer agent), we could represent that as an alternative in the TAEMS structure by by making it and "Get-Info-Address-From-Matchmaker" submethods of "Get-Info-Address" with a quality accumulation function of "q-max".

In addition to tasks, methods, and outcomes, TAEMS structures consist of interrelationships, also called non-local-effects, between tasks and methods. Interrelationships describe the way in which the outcomes of methods can affect other tasks or methods. For example, Figure 2 contains many **enables** relationships. An **enables** between task 1 and task 2 means that task 2 can not accrue any quality until task 1 has completed and the results of its execution are available. Another example of an interrelationship is **uses**, which is a relationship between a method and a resource. **Uses** means that the method requires a certain number of units of the resource to be available in order to execute. The exact number is specified in the parameters of the **uses** relationship. Another common interrelationships are **facilitates** between two methods, which means that the facilitated method can run faster, or achieve more quality, if the facilitating method has completed execution.

4 Graph Grammars

Graph grammars are a powerful tool used in a number of domains (Mullins & Rinderle 1991; Nagl 1979) to capture and characterize the underlying structural regularities. They offer a structured way to describe topological relationships between entities in a domain. Graph grammars are fundamentally similar to string grammars (Nagl 1979) with the difference lying in the productions. A *graph production* is a triple $p = (g_l, g_r, \mathcal{E})$ where (g_l is the subgraph to be replaced (left hand side) and (g_r is the subgraph to be inserted in its place in the host graph. $\mathcal{E}$) is the *embedding transformation*. A number of schemes for graph grammars have been proposed and the primary differences between them arise from the differences in the embedding schemes. Much of the traditional

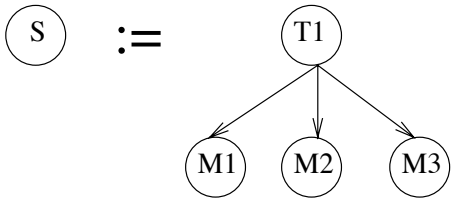
literature in graph grammars does not deal with attribute valued nodes and edges and stochastic productions. We need attributes to capture a number of other aspects of domain semantics in addition to the topological relationships between entities. Stochasticity in the productions adds more power to the modeling potential of these grammars by capturing aspects of uncertainty in the domain. Accordingly, we call our grammars Attribute Stochastic Graph Grammars (ASGGs).

Let a graph G = (V,E), where V is the set of vertices (also referred to as nodes) and E is the set of edges. Nodes and edges can have labels. An Attribute Stochastic Graph Grammar is defined as a 8-tuple $< \Sigma_n, A_n, \Sigma_t, A_t, \Delta, A_\Delta, S, P >$, where the nonterminal node alphabet (Σ_n), the terminal node alphabet Σ_t, and the edge-label alphabet (Δ) are finite, non-empty, mutually disjoint sets, A_n, A_t and A_Δ are the respective sets of attributes, S $\in \Sigma_n$ is the start label and P is a finite nonempty set of graph production rules (Sanfeliu & Fu 1983). A graph production is a 5-tuple $p_i = < g_l^i, g_r^i, \mathcal{E}, Pr(p_i), \delta >$ where $\sum Pr(p_j) = 1 \mid (p_j \in P$ and g_l^i, called the left hand side, is a single node, and g_r^i, called the right hand side, is a graph with a single root node. δ is a set of attributes that will be given to the root node of g_r^i. These attributes are passed on to the TAEMS task or method that the root node of g_r^i represents.

There are actually two different types of graph productions, both with the same format given above. A production, sometimes called a rule, is a *task rule* when the root node of the right hand side g_r^i, is a non-terminal node. The other case is that g_r^i is identical to g_l^i. This type or production is called an *attribute rule*, and it is a unitary rule. Attribute rules do not change nodes into something else, all they do is copy a new set of attributes, δ to the node g_r^i. Since the node of an attribute rule is not changed, the left hand side can be a terminal. In fact, these attribute rules are usually used to give new attributes to terminal nodes.

Let G' be a graph derived from S using P. Rewriting this graph involves what is called a LEARRE method (Mullins & Rinderle 1991): Locate a subgraph g' that is isomorphic to the left hand side, g_l^i of production $p_i \in$ P, establish the Embedding Area in G', Remove g' along with all the edges incident on it, Replace g' with g_r^i and Embed it to the host graph G' - g'. (Figure 3 shows a simple example)

The graphs we deal with are always directed, so the parents of a node are the set of nodes that have outgoing edges connecting them to the node in question. The children of a node are the nodes that are at the other end of an outgoing edge. The left hand side of a rule is always a single node, and the right hand side always has a single root node. The embedding rule will replace the single root node of the left hand side with the subgraph of the right hand side. The parent nodes of the old left hand side node will now point to the root node of the new subgraph. Any children of the left hand side node that are not present in the new subgraph are now disconnected, and thus are removed. If any nodes in



Embedding:
Connect T1 to parents(S)

Figure 3: Example of a Graph Grammar Rewriting

the right hand side graph are already present in the graph, those nodes inherit all links and attributes of the original leaves.

The grammar also specifies a start node, that will become the top level TAEMS task. The production list is searched for the set of productions whose left hand side node matches the start node. There can be multiple grammar productions with the same left hand side, provided that their probabilities sum to no 1.0. One of those productions is chosen randomly, based on the specified probability distributions. This process is repeated for every new node added to the structure. If no rule can be found, that node is a terminal, and thus a TAEMS method.

5 Dynamic Environments

The graph grammar defined in the previous section is quite capable of generating static task structures and simulating the domain problem solving component of an agent (Prasad 1996). However, in a dynamic environment changes may occur, such as resource failures or malevolent intrusions, that could cause the environment to behave differently from what is expected by the agent. For example, an agent has certain expectations about the cost and time for executing certain methods. If the method requires the use of a resource, these expectations will be based on the access characteristics of that resource. If a change in the environment causes a change in the access characteristics of the resource, then the agent's expectations about the cost and time required for that method could be incorrect. As an example, consider a method that involves reliable transfer of data over a network. If a link in that network fails, it may take a lot longer than the agent expected for the data transfer to succeed.

The point of all this is that if the generator is to simulate the environment properly, it must account for the fact that an agent's beliefs about the environ-

ment may not be correct. The generator does this by creating two views of a task structure, the *objective* view which represents the true quality, cost, and duration distributions of the methods, and the *subjective* view, which represents the agent's beliefs about the quality, cost, and duration distributions. In the data transfer example described in the previous paragraph, the objective view would have a higher average duration for the data transfer task than the subjective view, since the agent believes that the network is operating correctly, when in fact part of the network is down.

In addition to containing different quality, cost, and duration distributions, the objective and subjective views can differ in a multitude of other ways. First, the agent might not be aware of all of the task interrelationships. An agent might not know that a method is dependent on the state of a certain resource, it might be incorrect in its belief about how much of a certain resource is required, and it might not know that a method is affected by the results of another method executed by a different agent. Also, an agent might believe some relationships exist when in fact they do not. All of these things are represented by TAEMS non-local effects, also called task interrelationships. The objective view could contain a different set of interrelationships than the subjective view. The interrelationships present in the subjective view are not necessarily a subset of those in the objective view, since the agent might believe in interrelationships that do not exist.

The objective view can also contain entire task subtrees or methods that are not present in the subjective view. This would represent a way of solving a problem that the agent is unaware of, but the agent might become aware of, and able to execute, at a later time. Also, the tasks in the objective view could contain different quality accumulation functions, which determine how the quality in the subtasks is to be combined, than in the subjective view. For example, an agent might believe that both of a tasks subtasks must be executed in order for the task to accrue any quality, when in fact, either of the two subtasks executed alone would do the job.

The differences between the objective and subjective views can be used to simulate malevolent intrusions into the system. An outside force could modify or destroy one of the agents, or could affect the environment in a number of ways. If this happens, elements of the system will not behave as they are supposed to. For example, a new agent could be placed in the system whose only goal is to use up as many resources as possible. This agent could attempt to avoid detection by sending a false subjective task structure to the other agents when they want to coordinate. The intrusive agent's objective view would contain the method outcomes that consume many resources, while the subjective view that gets communicated to the real agents would contain method outcomes with little to no resource usage.

Since there are so many ways for the objective and subjective views to dif-

fer, the generator must provide ways for the problem designer to specify what goes in the objective view, and what goes in the subjective view. Once these views are generated, the agent is provided with a problem to solve, and a set of beliefs that it must work with. The problem then becomes one of scheduling, coordinating with other agents, and executing the tasks.

5.1 Reactions and Event Rules

In a dynamic environment, changes that affect the quality, cost, or duration of methods might occur at any time. If changes occur, the objective task structure would have to be modified while the agents are working. For example, if a network link all of a sudden fails, the duration of any method that sends data over that link would need to be increased, to reflect the fact that that data now has to find an alternate route to its destination. In addition to random environmental changes, the environment can be modified by the actions of the agents. For example, one agent might be able to repair a faulty network link, or a malevolent agent might actually cause failures to occur.

The generator can be used to model dynamic changes in the environment through the use of a special kind of graph grammar rule, called an event rule or reaction. An event rule is a 6-tuple $er_i = < g_l^i, \lambda, g_r^i, \mathcal{E}, Pr(p_i), \delta >$ where $\sum Pr(p_j) = 1 \mid (p_j \in P$ and g_l^i is a single node, and g_r^i is a graph with a single root node. $\mathcal{E}$ is the embedding transformation, which is the same as in section 4, and δ is a set of attributes. λ is an event. The reaction will not execute until that event occurs, and g_l^i exists in the task structure. One thing to note is that resources are actually special nodes in the TAEMS task structure with local state, so a resource node could be modified by an event rule. For a resource node, or any terminal node, g_l^i would equal g_r^i, and the effect of the event rule would come from the new set of attributes, δ

There are many different ways for an event to be triggered. Methods completing execution or methods achieving a quality, cost, or duration greater than or less than a certain threshold, can trigger an event. Also, a trigger could be a method's execution resulting in a certain outcome. In addition events can be set to trigger at a certain time. Since the generator knows in advance which events it has reactions for, it knows which events it cares about. Therefore, the generator can then place triggers for only those events that reactions exist for in the TAEMS task structures.

For example, consider a malevolent agent that purposely overloads a network connection by executing a method called Flood-Network. We would set up an event rule that looks like:

R = $< NetworkResource, FloodNetworkCompletes, NetworkResource, 1.0,$
$state = overloaded >$

Then, the generator will place a reaction trigger in the attributes of the Flood-Network method. This trigger will cause an event called "FloodNetworkCompletes" to be created when the method Flood-Network has finished executing. We could also have made the trigger go off when Flood-Network starts execution, or after Flood-Network as been executing for a certain length of time.

5.2 Simulating a Problem Solving component

Building a domain specific problem solving component can be a complex task, and such a component would have to be recreated for each new domain. These generator reactions and event rules can also be used to simulate the problem solving or learning components of an agent. The agent designer can use event rules to modify the subjective task structure of an agent, thus changing the agent's beliefs about the problem to be solved. It might take many rules to completely simulate the problem solving activities of an agent, however usually useful results can be obtained from only simulating some of an agents problem solving capabilities.

For example, consider a agent who tries to execute a method that sends data over a network. Initially, that agent will have an expected quality, cost, and duration distribution for that method that may look something like

```
Method: Send Data
Standard Outcome:
   Quality: 100% 10.0
   Duration: 100% 5.0
   Cost: 100% 1.0
```

This says that the agent believes that sending this data will always take 5 time units, at a monetary cost of 0, resulting in 10 units of a quality, which is an abstraction of the agent's progress toward its goals. Now, assume that the malevolent agent described in the previous section does its deed and floods the network. The resulting reaction and event rule would cause the Network resource to perform much worse than before. Now, if the Send-Data method is executed again, the duration would be greater than 5. A learning module of a problem solving component could recognize the discrepancy between the agent's beliefs and the actual results, and modify the expected duration distribution accordingly. However, if such a component is not available, the generator's event rules can be used to simulate this behavior. An event rule can be created with a trigger that goes off if the duration of Send-Data is greater than 5. The actual rule could simply change the duration distribution to reflect the greater uncertainty.

As another example, consider the information gathering agent, whose task structure was depicted in Figure 2. Lets assume that the agent attempts to ping the information source agent through the "Ping-Info-Agent1" method and fails (achieves a quality of 0 from outcome O2). This means that the information gathering agent was unable to contact the appropriate information source agent. One possible resolution to this problem is to ask another information gathering agent to see if it can respond to this query. Since this agent failed at processing this query, maybe it can contract out the query to another agent. In this case, we want to abort the ProcessQuery task structure and begin a new structure, ContractQuery. ContractQuery would be a task structure that represented the agent broadcasting the query to other agents, receiving bids, choosing a contractor, and finally delivering the end results computed by the contractor back to the interface agent. This can be represented by a reaction that triggers when Ping-Info-Agent1 results in outcome O2. The associated event rule would cause the current task structure, ProcessQuery, to abort, and a new task structure, ContractQuery, to be instantiated.

6 Generator Syntax

The generator operates on two types of files, a script file which tells it which task structures to generate when, and a set of grammar files which contain the graph grammar rules for generating an individual task structure. The generator outputs a log file, which records the actions taken by the generator, and a textual TAEMS file that contains the TAEMS task structure. When the generator is added as a component of an agent in our Java Agent Framework (Horling 1998), the generator also outputs the TAEMS structure as a Java class for use by other components of the agent.

In this section, we will create a grammar input file example to produce a task structure for a set of four information gathering agents. The first agent, the Interface Agent, has the job of interacting with the user. The user will input a query, and the Interface Agent will then ask a Task agent to do the work. The Task agent might need to contact a few Information agents to gather the data it needs to answer the query. The Information agents query a source of information, such as a database, to answer requests from the Task agent. These agents learn about the existence of each other through the services of a matchmaking agent, called the Broker agent. For simplicity, the Broker agent was not modeled, but it could easily be added. This system is an example of the Warren portfolio management architecture [11]. An example of a task structure that we want to produce with this grammar is given in Figure 4.

The graph grammar text file, which is case-sensitive, is parsed by the JavaCC Compiler Compiler from SUN Microsystems [11]. The input file consists of four different types of statements: task rules, attribute rules, the start symbol, and

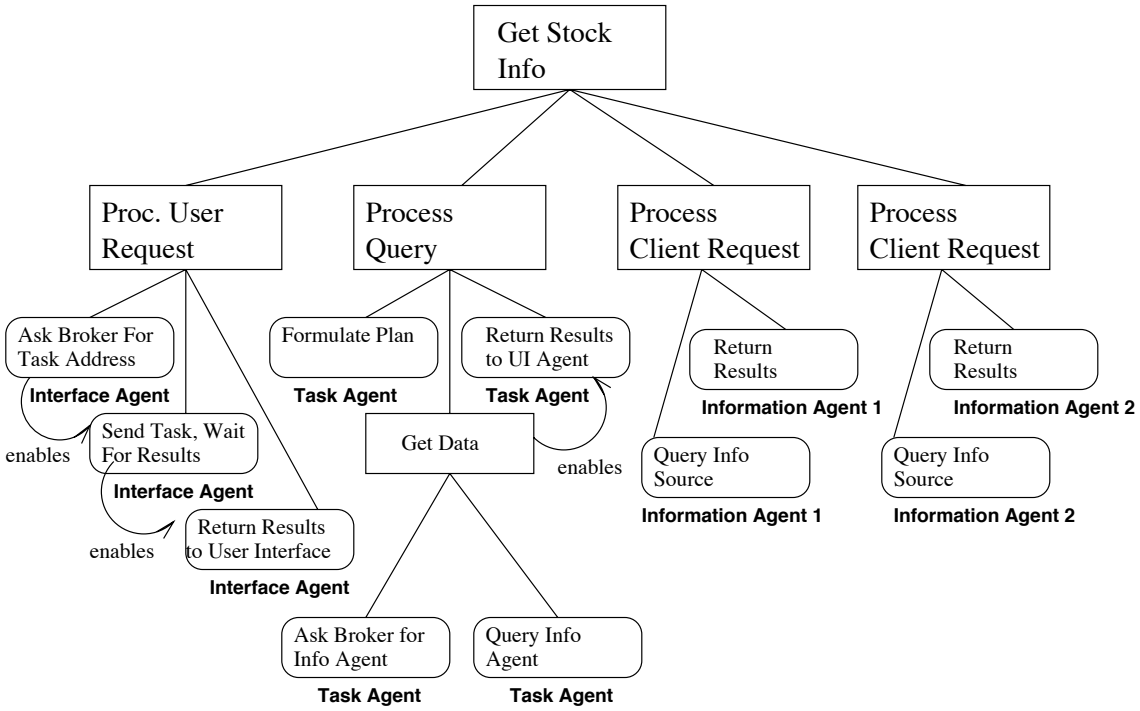


Figure 4: Example of a Warren task structure

extra TAEMS structures.

We will start our Warren grammar file by defining the start symbol. This is probably the easiest part of constructing a graph grammar. The start symbol will become the top-level task of the task structure. The format for a start symbol declaration is given below. In this section, the formats for grammar file entries will be written in the standard font, and the examples that will make up the actual Warren grammar input file will be given in the `typewriter` font.

(START-SYMBOL symbol)

```
(START-SYMBOL Get-Stock-Info)
```

Next, we need to define the structure of our task tree. A task rule will take a nonterminal node and add some subtasks to it. These subtasks are also nonterminal nodes which can be expanded through other task rules, or changed to terminal methods by the attribute rules described in the next section.

The format for a task rule is:

(TASKRULE lhs *label* rhs (*label1 label2*...) probability *p* count *integer* subjective attributes ((*attribute value*)...))

The required elements of a task rule are the left hand side (lhs) and the right hand side (rhs). If no probability is specified, the probability that the rule fires is 1.0. If multiple rules exist with the same left hand side, their probabilities need to sum to 1.0. If they don't, a warning will be output to the log file. The count field specifies how many times this rule can fire during one generation. This is used for recursively generated task structures that might have multiple nodes with the same name. The default count is infinity. The subjective keyword is used if you want the rule to fire only when constructing the subjective view. If the subjective keyword is present, the rule will not fire when constructing the objective view. The attributes allowed in the attribute section are listed below. The format for some of the non-trivial values (such as the non-local-effects) are given in a following section.

- qfn: Quality Accumulation Function (q-seq, q-max, q-min, q-sum, q-seq-sum, q-seq-min, q-seq-max, q-exactly-one, q-sigmoid)
- agents: Agent assigned to this task (can be inferred from subtasks)
- deadline: The time at which this task needs to be completed by
- earliest start time: The earliest time at which work on this task can commence.
- enables: This task enables another task. This will create a TAEMS NLE with this task as the enabling task.

- uses: This task uses some resource
- other TAEMS non-local effects, such as facilitates, hinders, or disables

Our Warren task structure will use the following task rules. Note the probability field of the Get-Data task rules. There is a fifty percent chance that the first Get-Data rule, which has the Task agent ask Info Agent 1 for the data, is used, and a fifty percent chance that the next rule, which has the Task agent ask Info Agent 2 for the data, is used.

```
(TASKRULE lhs Get-Stock-Info rhs (Process-User-Request Process-Query Process-Cl
ient-Request-1 Process-Client-Request-2) probability 1.0 attributes
((q-fn q_sum)))

; Interface Agent
(TASKRULE lhs Process-User-Request rhs (Ask-Broker-For-Task-Address Send-Task-W
ait-For-Results Display-Results-To-User) probability 1.0 attributes ((q-fn q_se
q_sum)))

; Task Agent
(TASKRULE lhs Process-Query rhs (Formulate-Plan Get-Data Return-Results-To-UI-A
gent) probability 1.0 attributes ((q-fn q_seq_sum)))

; 50% chance of Querying Info Agent 1 and 50% for querying Agent 2
(TASKRULE lhs Get-Data rhs (Ask-Broker-For-Info-Address Query-Info-Agent-1) pro
bability 0.5 attributes ((q-fn q_seq_sum) (enables Return-Results-To-UI-Agent))
)

(TASKRULE lhs Get-Data rhs (Ask-Broker-For-Info-Address Query-Info-Agent-2) pro
bability 0.5 attributes ((q-fn q_seq_sum) (enables Return-Results-To-UI-Agent))
)

; Information Agent 1
(TASKRULE lhs Process-Client-Request-1 rhs (Query-Info-Source-1 Return-Results) probability

; Info Agent 2
(TASKRULE lhs Process-Client-Request-2 rhs (Query-Info-Source-2 Return-Results) probability
```

Attribute rules are used to specify attributes for the terminal methods.

(ATTRIBUTES *method label* probability p subjective ((*attribute value*)...))

The probability field is optional, and the default value is 1.0. If there are multiple attribute rules for a method, their probabilities must sum to 1.0. The subjective field is also optional. If a subjective agent is set, these attributes will only apply to the subjective view of that agent. If this is the case, there

must be another attribute rule for this method that specifies the objective view attributes. By default, if subjective is not declared, the rule applies to the objective view. For example if we had the rules:

(ATTRIBUTES task1 attributes (attribute-set1)
(ATTRIBUTES task1 subjective attributes (attribute-set2))

then, the objective view of task1 would have attribute-set1, and the subjective view of task1 would have attribute-set2.

The types of attributes available are given below.

- agents: Which agent(s) are assigned to this task
- outcomes: quality, duration, and cost distributions
- reactions
- enables, facilitates, hinders, uses, limits

A TAEMS method might have multiple outcomes. An outcome consists of three distributions, quality, duration, and cost. A distribution is a list of numerical pairs. The first number in a pair is a percentage p, the second number is a value, v. The semantics are that the quality, duration, or cost will be equal to v, p percent of the time. All of the percentages in a distribution list should sum to 1.0. For example, to specify that the 25% of the time the quality will be 10, and 75% of the time the quality will be 5, one would write: (quality (0.25 10) (0.75 5)). The outcome attribute is a list of outcomes. There can be multiple outcomes, each of which has a certain probability of occurring. Multiple outcomes are used to create a subjective view that is different from the objective view. A subjective outcome that is different from the objective outcome can be defined with the subjective keyword. Note that the subjective keyword is optional. At least one outcome must not have this keyword, since at least one outcome must be defined for the objective view. If no subjective outcomes are defined, the subjective view is equal to the objective view.

(outcomes (density *percentage* subjective (duration *distribution*) (quality *distribution*) (cost *distribution*) ...))

Some of the attribute rules in our Warren grammar are shown below. Note that the Query-Info-Agent-1 rule has two outcomes entries, both representing the same actual outcome. The first outcome entry is for the subjective view, which is what the agent will believe the result of Query-Info-Agent-1 will be. The other is for the objective view, meaning that this is what the real result of Query-Info-Agent-1 will be. The result of this is that the Task agent believes that Query-Info-Agent-1 has a twenty-five percent chance of returning a quality

of 500, when in fact Query-Info-Agent-1 will always return a quality of 0, representing failure. This attribute rule is modeling a situation where Info-Agent-1 is malfunctioning, but the Task Agent is unaware.

```
(ATTRIBUTES Display-Results-To-User (
(agents Interface_Agent)
(outcomes ((duration (1.0 3)) (quality (1.0 10)) (cost (1.0 0))))))

(ATTRIBUTES Ask-Broker-For-Info-Address (
(agents Task_Agent)
(outcomes ((duration (1.0 12)) (quality (0.01 10) (0.99 0.1))  (cost (1
.0 0)))))

(ATTRIBUTES Query-Info-Agent-1 (
(agents Task_Agent)
(outcomes (subjective Task_Agent (duration (1.0 28)) (quality (.75 0.1)
 (.25 500)) (cost (1.0 0)))
((duration (1.0 28)) (quality (1.0 0.0)) (cost (1.0 0))))))

(ATTRIBUTES Query-Info-Agent-2 (
(agents Task_Agent)
(outcomes ((duration (1.0 28)) (quality (1.0 4)) (cost (1.0 0))))))

(ATTRIBUTES Return-Results-To-UI-Agent (
(agents Task_Agent)
(outcomes ((duration (1.0 8)) (quality (1.0 10)) (cost (1.0 0))))))

(ATTRIBUTES Return-Results-To-Agent-b (
(agents Information_Agent_2)
(outcomes ((duration (1.0 5)) (quality (1.0 5)) (cost (1.0 0))))))
```

Non-local-effects (NLEs) are specified as attributes of a rule. An NLE attribute has the following general format:

(*type* outcome *outcome label* (*affected method or resource*) ((subjective (from | to | none)) | objective | no-objective)((*distributions*)...))

The type must be one of the supported NLE types, currently enables, disables, facilitates, hinders, uses, or limits. The outcome field is optional, and is used when this NLE originates from a specific outcome of the method. By default, an NLE will be active no matter what outcome results from the execution of the method. If the outcome field is specified, the NLE will only be active if that specific outcome occurs. The distributions can be either cost (for a uses), delay (for all NLEs), duration (for a uses, facilitates, hinders), or quality (for facilitates, hinders). Some specific examples are given below.

```
(uses NetworkBandwidth (consumes (1.0 5)))
```

If we model the bandwidth of the network our Warren agents are operating in as a resource, a uses NLE like this will represent this method using five units of the resource.

```
(enables Query-Info-Agent)
```

This means that whatever method this attribute is assigned to enables the Query-Info-Agent method.

```
(hinders methodx (quality (1.0 0.5)) (duration (1.0 2)))
```

This means the associated method hinders methodx by causing methodx's quality to decrease by a factor of 2 and methodx's duration to increase by a factor of 2

Interrelationships can be in any of three different views, the objective view, the subjective view of the "from" tasks agent, and the subjective view of the "to" tasks agent. By default, an interrelationship is in all 3 views. The only exception to this is if the rule that the NLE definition is an attribute of is a subjective rule ((ATTRIBUTE blah subjective (...))), then by default it is NOT in the objective view.

Take the following example:

```
(ATTRIBUTES Return-Results subjective (
            ...
            (uses NetworkBandwidth (cost (1.0 5)))))

(ATTRIBUTES Return-Results (
             ...
            (uses NetworkBandwidth (cost (1.0 25)))))
```

By default, the (uses NetworkBandwidth 25) is present in the objective and subjective view of this agent. The (uses NetworkBandwidth 5) is present only in the subjective view. So, there are 2 uses relationships in the subjective view. To change the default behavior, you can use three keywords, placed just before (or after) the distribution definition:

- objective: This interrelationship will be put in the objective view
- no-objective: This interrelationship will not be put in the objective view
- (subjective (from | to | none)): This IR will be put in the specified subjective views.

For example, if we change the uses NetworkBandwidth 25 definition to: (uses NetworkBandwidth (subjective none) (cost (1.0 25))), then it will not be put in the subjective view, only the objective view. For an enables relationship, we might do: (enables blah (subjective from)), specifying that this relationship will be in the enabling agent's subjective view, and not the enabled agent's view. The enabling agent is called the "from" agent, and the enabled agent is the "to" agent.

The rest of the attribute rules present in our Warren example are given below. These rules all have non-local-effects as attributes.

```
(ATTRIBUTES Ask-Broker-For-Task-Address (
(agents Interface_Agent)
(outcomes ((duration (1.0 5)) (quality (1.0 10)) (cost (1.0 0))))
(enables Send-Task-Wait-For-Results)
(enables Formulate-Plan)))

(ATTRIBUTES Send-Task-Wait-For-Results (
(agents Interface_Agent)
(outcomes ((duration (1.0 127)) (quality (.05 35) (.75 50) (.20 75)) (c
ost (1.0 0))))
(enables Display-Results-To-User)))

(ATTRIBUTES Formulate-Plan (
(agents Task_Agent)
(outcomes ((duration (1.0 5)) (quality (1.0 10)) (cost (1.0 0))))
(enables Get-Data)))

(ATTRIBUTES Ask-Broker-For-Info-Address (
(agents Task_Agent)
(outcomes ((duration (1.0 12)) (quality (0.01 10) (0.99 0.1))  (cost (1
.0 0))))
(enables Query-Info-Agent)))

(ATTRIBUTES Query-Info-Source-1 (
(agents Information_Agent_1)
(outcomes (subjective Information_Agent_1 (duration (1.0 16)) (quality
(.50 50) (.5 100)) (cost (1.0 0)))
((duration (1.0 16)) (quality (1.0 0.0)) (cost (1.0 0))))
(enables Return-Results-To-Agent)))

(ATTRIBUTES Return-Results-To-Agent (
(agents Information_Agent_1)
(outcomes (subjective Information_Agent_1 (duration (1.0 5)) (quality (
1.0 5)) (cost (1.0 0)))
```

```
((duration (1.0 5)) (quality (1.0 0)) (cost (1.0 0))))))

(ATTRIBUTES Query-Info-Source-2 (
(agents Information_Agent_2)
(outcomes ((duration (0.8 16) (0.2 32)) (quality (1.0 40)) (cost (1.0 0
)))
(subjective Information_Agent_2 (duration (1.0 16)) (quality (1.0 40))
(cost (1.0 0))))
(enables Return-Results-To-Agent-b)))
```

Reactions are specified as attributes of a task or method, and thus specified in the grammar file as attributes of a rule. There are two different types of triggers that a reaction can have. First, the reaction could fire if a method completes execution and results in a specific outcome. If the outcome that occurs matches the label specified in the reaction, the reaction will fire and an event will be created. The second type of reaction will fire if a specific cost, duration, or quality value is reached.

(reactions (label *label* (*trigger-type trigger-value*) (view (sub | obj)* (monitoring-interval *interval*))

The trigger types are:

- (*global* – *timenum*). Will fire after num time units have elapsed
- (*outcome* – *triggeroutcome* – *label*). Will fire if the method's execution results in this outcome
- (*quality* – *minnum*). Will fire if the quality is less than num
- (*quality* – *maxnum*). Will fire if the quality is greater than num
- (*duration* – *minnum*). Will fire if the duration is less than num
- (*duration* – *maxnum*). Will fire if the duration is greater than num
- (*cost* – *minnum*). Will fire if the cost is less than num
- (*cost* – *maxnum*). Will fire if the cost is greater than num

The view field specifies which task structures are to be affected by this rule. The options are "sub" for the subjective view and "obj" for the objective view. It is possible to specify both the objective and subjective view in the view field, this is the default. By default, these reactions only check the quality, cost, and duration values against the trigger conditions after execution has completed. However, reactions can monitor the quality, cost, and duration of a method while that method is executing. In order to do this, a monitoring interval which specifies how often to check the trigger condition needs to be set. For example,

you might want the reaction to fire if the quality is still less than 2 after 10 time units of execution have passed. To do this, you would use the monitoring-interval keyword, and set the interval to 10.

Two examples of reactions in are given below. The first reaction will fire if the method "Ask-Broker-For-Info-Address" method fails, where failure is defined as completing with a quality less than 2.

```
(ATTRIBUTES Ask-Broker-For-Info-Address (
        (q-fn q_max)
        (agents Task_Agent)
        (outcomes ((duration (1.0 12)) (quality (0.01 10) (0.99 0.1))
           (cost (1.0 0))))
        (reactions (label BadBroker (quality-min 2)))))

(EVENTRULE lhs Ask-Broker-For-Info-Address (view obj sub) event BadBroker rhs (Ask-Broker-1-
  probability 1.0 attributes((q-fn q_seq_sum) (agents Task_Agent)))
```

The failure of "Ask-Broker-For-Info-Address" means that the Task Agent was unable to discover the location of an appropriate Information Agent from the Broker Agent. This means that the Task Agent's job of processing the query can not be completed. A dynamic Task Agent, complete with a domain problem solver, might be able to figure out another way to locate an Information Agent. We can use the generator and its event rules to simulate this domain problem solving action.

If the "Ask-Broker-For-Info-Address" method fails, a BadBroker event is created. This event will be matched with the above event rule. This causes the Ask-Broker-For-Info-Address method to be replaced by an Ask-Broker-For-Info-Address task which has three subtasks, the methods Ask-Broker-1-For-Address, Ask-Broker-2-For-Address, and Choose-Address. The "(view obj sub)" parameter tells the generator to modify both the objective and subjective views. Modifying the objective view tells the simulator about the change, and modifying the subjective view tells the agent about the change. This reaction simulates problem solving component querying two different Broker Agents for an address and choosing one.

We can also use reactions and event rules in a different way, to simulate a change in the environment. In this case, we would want to modify the objective view, which represents what is true, and not the subjective views of the agents, which represent what the agents believe to be true. For example, assume that at time 500, we want to simulate an intrusion in the system that results in one of the Information Agents becoming compromised and sending out incorrect information. We set up another reaction, with a global-time trigger set to 500.

This reaction will fire off an event labeled Intrusion, which makes the quality of the Information Agent's query zero. The parameter "(view obj)" means that only the objective view will be changed and not the subjective view. Thus, the Task Agent will not know that Query-Info-Source-2 is doomed to failure. Discovering this fact would be a job for a detection and diagnosis component, with help from a learning component.

```
(ATTRIBUTES Query-Info-Source-2 (
        (q-fn q_max)
        (agents Task_Agent)
        (outcomes ((duration (1.0 16)) (quality (1.0 10)) (cost (1.0 0))))
        (reactions (label Intrusion (global-time 500)))))

(EVENTRULE lhs Query-Info-Source-2 (view obj) event Intrusion rhs (Query-Info-Source-2)
  probability 1.0 attributes(
(outcomes ((duration (1.0 16)) (quality (1.0 0)) (cost (1.0 0))))))
```

7 Script Input File

The script input file tells the generator when to generate each task structure. Structures can be generated periodically or aperiodically, an infinite number of times, or a set number of times. The task structures to be generates can be chosen randomly from a set of structures. The format for the script file is given below.

GRAMMAR: (DETERMINISTIC | NONDETERMINISTIC) START: *value*
TBTS: *value* COUNT: *value* PROBABILITY: *value*
FILE: *grammar input file name* PROBABILITY: *value*

A deterministic entry will only specify one grammar input file, and have all probability fields automatically set to the default of 1.0. The START field specifies at what time step to begin the first generation of this task structure on. The default value is time step 0. The TBTS field stands for "time between task structures", and specifies the interval between successive generations of the task structure. TBTS is a required field, with no default value. The COUNT field says how many times to generate this task structure, and it has a default value of infinity. For example, if the start value is 10, the time between successive task structures (TBTS) is 10, and count is 5, then the task structure will be generated and presented to the appropriate agents on time step 10, 20, 30, 40, and 50.

A non-deterministic entry specifies a set of grammar files that can be used for generation. The probability field on the GRAMMAR line gives the probability that any task structure from this set is generated. For example, with a start of 10, tbts of 10, count of 5, and a probability of 0.5, there will be a fifty

percent chance that a task structure is generated from the set at time 10, and a fifty percent chance of a structure being generated from the set at time 20, and so on. Following the GRAMMAR line, comes a set of FILE lines. Each of these lines gives the name of one grammar input file. The probability fields on these lines give the relative probabilities that this file will be used for generation as opposed to the other files. All of these fields must sum to 1.0. For example, the following statement will generate one task structure at time 10, with a 50 percent chance of generating from grammar1, 25 percent chance of generating from grammar 2, and a 25 percent chance of generating from grammar3.

```
GRAMMAR: NONDETERMINISTIC START: 10  TBTS: 10  COUNT: 1
  FILE: grammar1 PROBABILITY: 0.5
  FILE: grammar2 PROBABILITY: 0.25
  FILE: grammar3 PROBABILITY: 0.25
```

8 Conclusions and Future Work

The grammar generator in an interesting component of a multi-agent system that can be used in many different capacities in order to make system design and implementation easier. First, the generator serves as a model of the environment, providing task structures to the agents and to the simulation framework. The dichotomy of the objective and subjective views allows the simulation of uncertainty and malevolent intrusions in the environment. The new dynamic components, the event rules and reactions, allow the simulated environment to change over time. In addition, these event rules can be used to simulate agent learning and problem solving. Therefore, the system designer need not develop a domain specific problem solver for each new domain in order to study a multi-agent system operating in that domain.

The generator is fully integrated with our MASS simulator, however it can also be run in a stand-alone mode. In this mode, the generator simply outputs task structures that can be used by another component. This stand-alone mode has been used by scheduling researchers trying out new scheduling technologies on the task structures. Of course, with no simulation environment, reactions and the dynamic portions of the generator are unaccessible.

The TAEMS grammar generator has seen wide use in our MASS simulator environment, modeling problems from an Intelligent Home domain (Lesser et al., 1998), a robot transportation domain, and a financial portfolio management domain. In addition, the reaction and event rule mechanisms will begin to see more wide spread use when more survivability issues are investigated. In addition, the generator and simulator were used as tools in a course project for a multi-agent problem solving seminar taught here at UMASS. Also, the graph

grammars are being investigated for use as input to a organizational designer.

References

[1] K.S. Decker and V.R. Lesser 1993. "Quantative Modeling of Complex Environments" In *International Journal of Intelligent Systems in Accounting, Finance, and Management. Special Issue on Mathematical and Computational Models and Characteristics of Agent Behavior*, Volume 2, pp. 215-234.

[2] K.S. Decker 1995. *Environment Centered Analysis and Design of Coordination Mechanisms.* Ph.D. Dissertation, University of Massachusetts, Amherst.

[3] K.S. Decker, A. Pannu, K. Sycara, and M. Williamson. "Designing Behaviors for Information Agents", in *Proceedings of the First International Conference on Autonomous Agents*(AGENTS-97), Feb. 1997.

[4] B. Horling 1998. "A Reusable Component Architecture for Agent Construction," Masters Thesis, University of Massachusetts, Amherst.

[5] V. Lesser, M. Atighetchi, B. Benyo, B. Horling, A. Raja, R. Vincent, T. Wagner, P. Xuan, S. XQ. Zhang, "Multi-Agent System for Intelligent Environment Control". In Technical Report TR-98-40, 1998, University of Massachusetts.

[6] S. Mullins and J.R. Rinderle 1991. "Grammatical Approaches to Engineering Design", part i. *Research in Engineering Design* 2:121-135

[7] M. Nagl 1979. "A Tutorial and Bibliographic Survey on Graph Grammars". In V. Claus, H. Ehrig, and G. Rosenberg, eds. *Graph Grammars and their Application to Computer Science and Biology*, LNCS 73. Berlin: Springer-Verlag, pg 70-126.

[8] M.V. Nagendra Prasad, K.S. Decker, A. Garvey, V. Lesser. "Exploring Organizational Designs with TAEMS: A Case Study of Distributed Data Processing", ???

[9] A. Sanfeliu, and K.S. Fu 1983. "Tree-graph Grammars for Pattern Recognition". In Ehrig, H.; Nagl, M.; and Rozenberg, G., eds., *Graph Grammars and their Application to Computer Science,* LNCS 153. Berlin: Springer-Verlag. 349-368.

[10] R. Vincent, B. Horling, T. Wagner, and V. Lesser, 1998. "Survivability Simulator for Multi-Agent Adaptive Coordination", In *International Conference on Web-Based Modeling and Simulation*, San Diego, CA

[11] The Java Compiler Compiler (JavaCC), Sun Microsystems. http://www.sun.com/suntest/products/JavaCC/index.html