

Design and Analysis of the MinneTAC-03 Supply-Chain Trading Agent

Wolfgang Ketter, Elena Kryzhnyaya, Steven Damer, Colin McMillen,
Amrudin Agovic, John Collins, Maria Gini
Dept. of Computer Science and Engineering,
University of Minnesota

Abstract

MinneTAC is an agent designed to compete in the Supply-Chain Trading Agent Competition [1]. It is also designed to support the needs of a group of researchers, each of whom is interested in different decision problems related to the competition scenario. The design of MinneTAC breaks out each basic behavior into a separate, configurable component. Dependencies between components are almost non-existent. This design allows each user to focus on a single problem and work independently, and it allows multiple users to tackle the same problem in different ways. This paper describes the design of MinneTAC and evaluates its effectiveness in support of our research agenda, and in its competitiveness in the TAC-SCM game environment. We also describe two sales strategies used by MinneTAC. Both strategies estimate, as the game progresses, the probability of receiving a customer order for different prices and compute the expected profit. Offers are made to maximize the expected profit on each order. The main difference between the two strategies is in how the probability of receiving an order and the offer prices are computed. The first strategy works well in high-demand games, the second was developed to improve performance in low-demand games. We empirically analyze the effect of the discount given by suppliers on orders received the first day of the game, and we show that in high-demand games there is a strong correlation between the offers an agent receives from suppliers on the first day of the game and the agent's performance in the game.

1 Introduction

One of the more compelling application areas for autonomous agents is in electronic commerce. Decisions can be relatively clear-cut (buy or sell, set a price, submit a bid, award bids, etc.), and communications among agents and between agents and their environments can be constrained and highly scripted.

One way to drive development and understanding in complex domains is to hold competitions. If we don't yet understand how to build an automated economic agent that will operate successfully in open, real-world economic environments, we can create slightly more constrained environments and carry out our competitions in those environments. An example of such an environment is the Supply-Chain Management Trading Agent Competition [1] (TAC SCM), which engages agents in simultaneous buying, selling, production scheduling, and inventory management problems.

In this paper we describe the design of our agent for TAC SCM 2003, and analyze its sales strategies. We have attempted to respond both to the challenges of the game scenario as well as to the need to support multiple relatively independent research efforts that are focused on meeting one or more of those challenges. We evaluate the success of our design both in terms of the competitiveness of the agents that have been implemented with it, and in terms of its ability to support our research agenda. We also describe two sales strategies used by our agent, and analyze their performance in different games. We show how the start-effect caused by the large discount given by suppliers on orders made on the first day, coupled with the random order in which agent requests are considered, affects the outcome of the game.

2 Overview of the TAC SCM game

In a TAC SCM game, each of the competing agents plays the part of a manufacturer of personal computers. In an instance of a TAC SCM game six autonomous agents compete with each other in a procurement market for computer components, and in a sales market for customers, as shown in Figure 1. Each agent is self-interested and tries to maximize profits, while competing with the other agents for customer orders. The simulation takes place over 220 virtual days, each lasting fifteen seconds of real time. Each agent starts with no inventory and an empty bank account, and must borrow (and pay interest) to build up an initial parts inventory before it can begin assembling and shipping computers. The agent with the largest bank balance at the end of the game wins. Agents earn money by selling computers they assemble out of parts purchased from suppliers.

A Component Catalog and Bill of Materials are sent to each agent at the beginning of the game. The Component Catalog lists each component, along with its base price and a list of suppliers who can produce it. Each component is produced by one or two suppliers; each supplier provides two different types of components. The Bill of Materials lists 16 different combinations of components that can be assembled into PCs. Each of these computer types is identified uniquely by a stock keeping unit number. Each computer type is assigned a number of processing cycles that determines how much time it takes to assemble that type of computer from raw materials. These PCs are the finished goods of the TAC SCM supply chain.

To obtain parts, an agent must send a *request for quotes* (RFQ) to an appropriate *supplier*. Each RFQ specifies a component type, a quantity, and a due date. The next day, the agent will receive a response to each request. Suppliers respond by evaluating each RFQ to determine how many components they can deliver on the requested due date and how long it would take to produce all the components requested, considering the outstanding orders they have committed to and the RFQs they have already responded to this turn. If the supplier can produce the desired quantity on time, it responds with an offer that contains the price of the supplies. If not, the supplier responds with two offers: (1) an earliest complete offer with a revised due date and a price, and (2) a partial offer with a revised quantity and a price. The agent can accept either of these alternative offers, or reject both. Suppliers may deliver late, due to randomness in their production capacities. If the supplier has excess capacity, the price will be discounted; discounted prices may be as low as 50% of the base price.

Once an agent has enough parts to assemble computers, it must schedule the assembly tasks in its production facility. Each computer model requires a specified number of assembly cycles, and the assembly capacity of each agent is limited. Assembled computers are added to the agent's finished-goods inventory, and may be shipped to customers to satisfy outstanding orders.

Every day each agent receives a set of RFQs from potential *customers*. Each customer RFQ specifies the type of computers requested, along with quantity, due date, reserve price, and late penalty. Each agent may choose to bid on some or all of the day's RFQs. Customers accept the lowest bid that is at or below their reserve price, and notify the agent the following day.

The agent must ship customer orders on time, or pay a penalty for each day an order is late. If a product is not shipped within five days of the due date the order is cancelled, the agent receives no payment, and no further penalties accrue.

3 The design of MinneTAC-03

Given the scenario outlined in the previous section, an agent must manage several major variables:

Raw Materials inventory and future modifications due to outstanding supplier orders and the production schedule,

Finished Goods inventory and future modifications due to outstanding customer orders and the production schedule,

Production converts raw materials into finished goods according to a schedule determined by the agent,

Bank account, debited when suppliers ship parts and for interest due, and credited when finished goods are shipped to customers and for interest paid, and

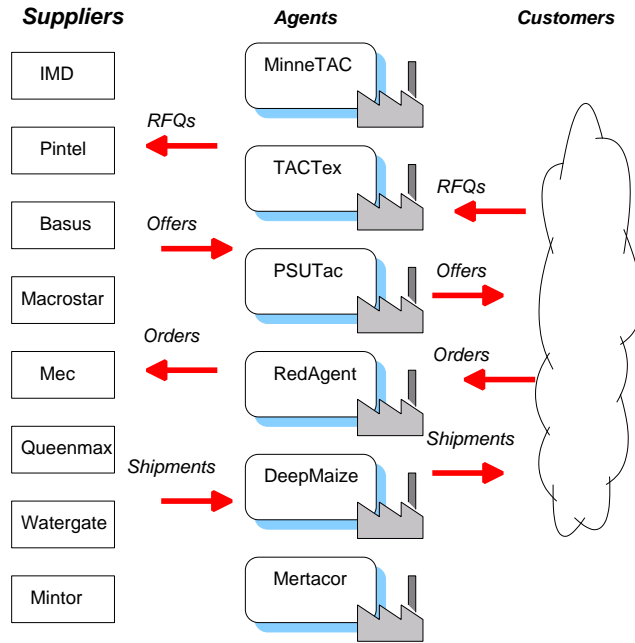


Figure 1: Schematic overview of a typical TAC SCM game scenario.

Time, because behavior at the beginning and end of the game is typically different from the steady-state behavior; for example, since inventory has no residual value at the end of the game, there is a strong motivation to run it down to zero.

Managing these variables requires the agent to make a number of decisions each day. These include setting prices and bidding on customer RFQs, issuing supplier RFQs, evaluating supplier offers and issuing orders to suppliers, shipping product to customers, and scheduling the production facility.

3.1 Design challenges

In addition to the design challenges presented by the TAC SCM problem domain, our research needs present an additional set of issues.

For example, our design must support multiple independent developers pursuing their own lines of research. The TAC SCM scenario presents a number of relatively independent decision problems, and there are many possible approaches to solving them. Our design must make it relatively easy for a researcher to focus on a particular subproblem without having to worry about getting a whole agent to work correctly. We also need to be able to configure agents with different combinations of decision process implementations.

We expect to continue participating in TAC SCM over several years, and we want to avoid redesign and re-implementation over that time, even though we expect significant details of the game scenario, as well as the communication protocol between the agent and the game server, to change from one year to the next.

Decision processes may involve somewhat arbitrary parameters, and their interactions and the sensitivity of agent performance to the settings of those parameters may not be well-defined. This is true even in cases where the agent is designed specifically to minimize the number of such parameters by use of optimization methods [13].

Experimental research requires data. The TAC SCM game server keeps data from each game played, which may be used to understand and compare the performance of competing agents. However, it is also necessary to integrate game data with information about the agent's internal state during the game, in order to understand the detailed performance of agent decision processes. This suggests a need for a data logging capability that can be easily configured to extract needed data from a running agent, while keeping the size of log files under control.

To address these design challenges, we follow a component-oriented approach [20]. The idea is to provide an infrastructure that manages data and interactions with the game server, allowing individual researchers to encapsulate agent decision problems within the bounds of individual components that have minimal dependencies among themselves. Two pieces of software form the foundation of MinneTAC: the Avalon component framework, and the “dummy agent” distributed by the TAC SCM game organizers. Avalon provides the standards and tools to build components and configure working agents from collections of individual components, and the dummy agent handles interaction with the game server.

3.2 A brief overview of Avalon

Apache Avalon [14] is a general-purpose component framework. It is widely used, primarily as a foundation for middleware and server software, such as the OpenORB CORBA implementation (OpenORB.sourceforge.net) but its use in the implementation of autonomous agents is rare. It does not provide the “classic” facilities for agent design, such as knowledge representation, inter-agent communication, reasoning facilities, or a planning infrastructure. Instead, it provides a means to build complex, robust systems from sets of role-based, configurable components. This satisfies a primary goal of MinneTAC, allowing researchers to work independently on individual decision problems with minimal need for detailed coordination with each other.

Avalon components are independent entities, in the sense that they typically have very few dependencies on each other, and minimal, well-defined dependencies on the Avalon framework itself. Components are coarse-grained entities, typically composed of a number of classes. Control inversion puts primary control in the Avalon “container”, which loads components, sets up logfiles, configures the components, and starts any components that run independent threads. Most components are passive entities, waiting for specific events to trigger their behaviors.

Each Avalon component is designed to fulfill a specific *role*, and an Avalon system is defined as a set of these roles. A role has a name, a set of responsibilities, and a well-defined interface. For most components, that interface is just a subset of the basic Avalon component interfaces:

Configurable. The component can be configured, by passing a portion of a configuration tree extracted from an XML configuration file.

Serviceable. The component uses (and depends on) other components. This is done through a ServiceManager that can look up and return references to other components based on role names.

Initializable. The component needs to be initialized, possibly by allocating some resources.

Disposable. The components must clean up allocated resources before being removed.

Startable. The component runs a thread of control.

An Avalon application, then, is composed of the Avalon infrastructure, the specified components, and a “container” that reads the configuration files and starts the process running. The configuration files specify the roles, the classes that satisfy those roles, and specific configuration parameters for those classes. The container reads the configuration files, loads the specified classes, and invokes the Avalon interfaces in order. Often, at least one component is Startable, and drives the ongoing behavior of the system. Avalon handles logging and the management of logfiles, and can be called on to handle resource management tasks while the system runs.

3.3 MinneTAC architecture

Figure 2 is a high-level view of the architecture of MinneTAC. It is conceptually simple: The Avalon container, and (at least) 7 components. All data that must be shared among components is kept in the Repository, which acts as a blackboard [5]. The components themselves are identified by their roles; in several cases multiple components have been built to fill those roles. It is an explicit goal of this architecture to minimize couplings between the components. Ideally, each component depends only on Avalon and the Repository.

The MinneTAC agent opens three configuration files when it starts. These are handed to the Avalon infrastructure, which reads and processes them. The system configuration file specifies the set of roles that make up the system.

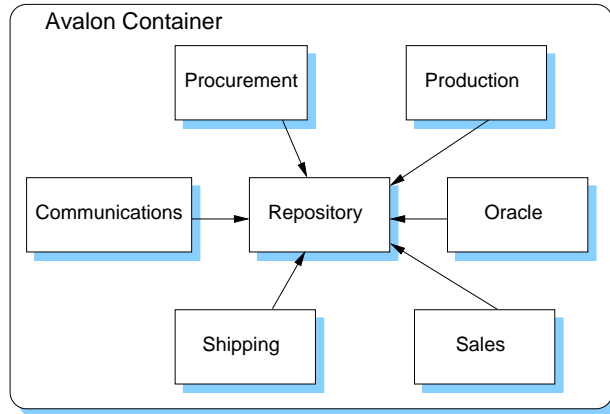


Figure 2: MinneTAC Architecture. Arrows indicate API dependencies.

Each role has a name, and a class that implements a subset of the Avalon component interfaces. The component configuration file specifies runtime configuration options for each component. For example, the Sales component may have a parameter that controls the default level of overcommitment of its existing inventory when it makes customer offers. The log configuration file controls the names and locations of log files that are produced by the running agent, the general format of log entries, and for each component, the level of detail to be logged.

3.3.1 Events.

A TAC Agent is basically a “reactive system” in the sense that it responds to events coming from the game server. These events are in the form of messages that inform the agent of changes to the state of the world: Customer RFQs and orders, supplier offers and shipments, etc. The game is designed so that each simulated day involves a single exchange of messages; a set of messages sent from the game server to the agent, and a set returned by the agent back to the server. For example, from the standpoint of the agent, each day’s incoming messages includes the set of customer RFQs for the day, and the return set of messages includes the agent’s bids for those RFQs.

More specifically, Figure 3 shows the communication activity for a game day. The general pattern is that the game server sends out a set of messages representing supplier and customer activity, as well as inventory and bank-account status data, the agent deliberates for some time, and then the agent responds with a set of messages that respond to the customer RFQs, and supplier offers for the current day. The agent must also specify the production and shipping schedules for the following day.

As shown in Figure 3, the agent does not need to react to individual messages from the server. Instead, it waits until after all the day’s messages have been received, and then considers all of them together. In fact, there is one additional end-of-data message not shown in the figure, which contains no data but simply tells the agent that the day’s input messages are complete. MinneTAC handles all data messages by storing them in the Repository. When the end-of-data message is handled by the Repository, it notifies the other components that the day’s data input is complete. Components use this notification as the signal to perform their deliberations and compose the day’s return messages. In this interaction, the Repository acts as a Subject and the other components as Observers in the *Observer* pattern [8].

Other events available to all components include a “start-of-game” event, to signal that the game parameters are available and that a new game is starting, and a “market-data-available” event to signal the components that new market summary data is available. This is necessary because market summary data is not provided every day.

3.3.2 Evaluators.

As we stated in Section 3.3, a goal of the MinneTAC design is to minimize coupling between the various components. How, then, do they communicate? One possible approach is the one used by the RedAgent team at McGill, in which the components communicate through internal auction-based markets. Our approach is to use *evaluations* that are

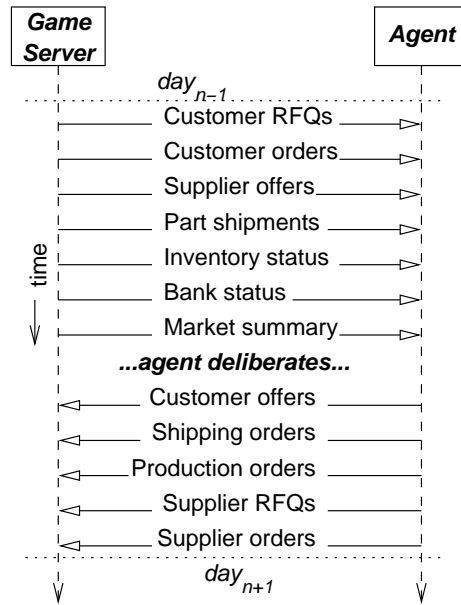


Figure 3: One day of communications activity between the game server and an agent.

attached to the various data elements in the Repository. The general idea is that when a component needs to make a decision, it will inspect the available data and run some utility-maximizing function. The available data consists of any data it maintains internally, and the data in the repository. Any data reductions or analyses that are performed on Repository data can be encapsulated in the form of Evaluations, and added back to the repository, where they will be visible to the other components.

In order to make Evaluations readily available to components, they are attached to the data elements from which they are derived. This is done by making all the major data elements in the Repository be subtypes of *Evaluable*. As shown in Figure 4, each *Evaluable* can have some number of associated Evaluations. Each Evaluation has a *type*, which is just a name, along with a value. Also associated with each *Evaluable* is an *EvaluationFactory*, which is responsible for filling in Evaluations when they are requested. It does this by inspecting the class of the *Evaluable* and the type-name of the requested Evaluation, and invoking the *evaluate* method on the associated *Evaluator*. Evaluators can implement a simple type of back-chaining by specifying other Evaluation types that must exist as preconditions before they are able to produce their results. Most evaluators are bundled with their respective components, and are registered with the Repository when the component is configured. To avoid infinite regress, cycles are not allowed in the *preconditions* relation among Evaluators.

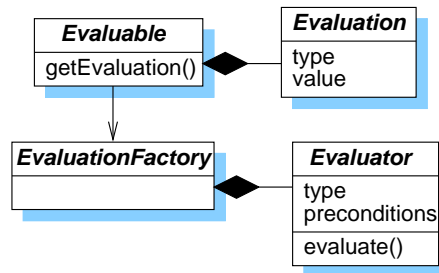


Figure 4: Evaluables, Evaluations, and Evaluators.

Figure 5 shows a simple example of some *Evaluable* instances and a set of *Evaluations* that might be associated with them. The *price* evaluation could be supplied by the Sales component, and it might require parts cost information

from the Procurement component as well as an estimate of current market conditions from the Oracle component. The *profit* evaluation would need parts cost information and *price*. The *sort-by-profit* evaluation would need the *profit* evaluations on the individual RFQs.

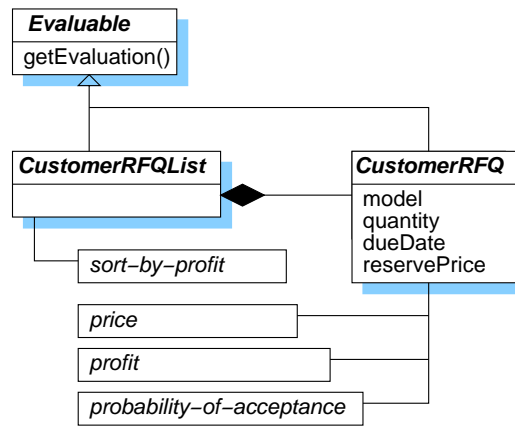


Figure 5: RFQ evaluation example.

3.4 MinneTAC components

The MinneTAC agent consists of 7 components. We describe these components and their responsibilities briefly here, and then provide more detail on the two “core” components, the Repository and the Communications component.

Repository is the unifying element of the MinneTAC design, the one component that is visible to the other components. It serves as an internal database, maintains the state of the system, and notifies other components of changes in state. All other activity is driven by these state changes. It also provides the core elements of the Evaluation subsystem.

Communications handles communication with the game server. This includes joining games, acquiring initial game parameters, importing start-of-game and daily data into the Repository, and retrieving agent decisions from the Repository for communication back to the game server.

Procurement procures parts. It may build and maintain target inventory levels, it may attempt to procure parts to meet customer orders, or it may use some other decision process. It must issue RFQs to suppliers and decide whether to accept offers that are returned.

Production schedules the manufacturing facility. It may build and maintain target finished goods inventory levels, or it may build only to meet existing customer orders.

Sales makes offers in response to customer RFQs. It must decide, for each RFQ, whether to bid and what price to quote, based on available and predicted inventories and current market conditions. A sophisticated Sales component might attempt to predict the probability of order acceptance in order to maximize profits.

Shipping ships product to customers. In general, there is a benefit in shipping product as late as possible, because this gives the agent an opportunity to minimize penalties for late deliveries. Late deliveries can happen, for example, if predicted inventories do not materialize due to late supplier shipments.

Oracle predicts future demand and availability.

3.4.1 Repository.

As stated earlier, the Repository is the one component that is visible to all the other components. The Communications component deposits new incoming messages into the Repository at the beginning of a day, waits for the decision processes to complete, and retrieves the agent’s responses from the Repository at the end of the day. Other components are notified when they must make their decisions. To perform their evaluations, they retrieve data from the repository, and record their decisions in the repository.

Figure 6 shows the state transitions and associated events in the Repository. Because the composition of the MinneTAC system is determined by configuration files, there is no fixed sequence in which components receive event notifications. When a component receives the *data-available* event, it is expected to perform whatever evaluations are necessary and record the results in the form of Evaluation instances and outgoing messages.

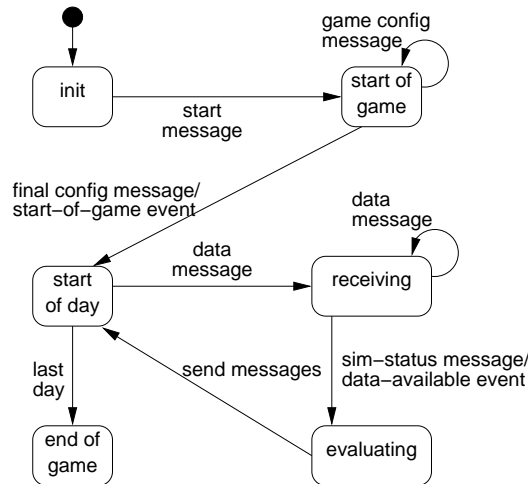


Figure 6: States and transitions in the Repository component.

Whenever there are data dependencies among components (for example, the Sales component might want to see the current day’s supplier orders before deciding whether to commit to an RFQ), an Evaluation is requested. The Repository’s EvaluationFactory is asked for any necessary Evaluations that have not already been computed and stored on their respective Evaluables. It responds by looking up and invoking the associated Evaluators as necessary. The existence of preconditions in Evaluator definitions can cause this to happen recursively.

The Repository plays the part of the Blackboard in the *Blackboard* pattern [5], and the remainder of the components, other than the Communications component, act as Knowledge Sources. However, the Control element of the Blackboard pattern is replaced by the Evaluable/Evaluator mechanism.

3.4.2 Communications.

In order to participate in a trading game, the agent must be able to communicate with the game server. The basic communication behaviors are provided in a *dummy agent*, provided by the game organizers. One way to construct an agent for the TAC game is to use the communication elements in the dummy agent code directly. However, our team felt that this was a risky approach, since changes to the communication protocol could ripple through to changes in the dummy agent, necessitating new rounds of code-extraction and disruptions to our own existing code. Our approach is therefore to simply “wrap” the entire dummy agent with an Avalon component that has responsibility for communications with the game server. The Communications component acts as an *Adapter* in the sense of [8]. We show this approach schematically in Figure 7. This way, we avoid modifying the dummy agent, and we are always able to use the latest version of the dummy agent by downloading it and importing it into our environment.

In most MinneTAC configurations, the Communications component is the only component that runs a thread; all the others react to events that originate ultimately from interactions with the game server.

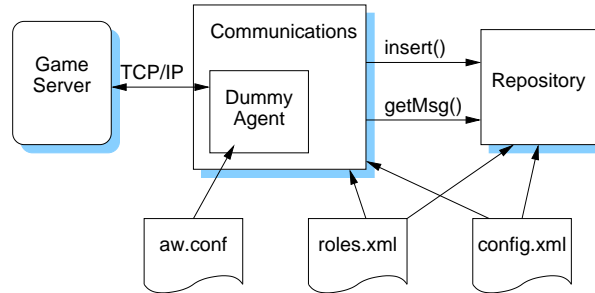


Figure 7: Communications component wraps the Dummy Agent.

3.5 Design evaluation

We evaluate the success of the MinneTAC design by asking two questions: (1) Does the agent perform well, and to what extent does the design affect agent performance? (2) Does the design meet the “usability” challenges described in Section 3.1?

3.5.1 Performance

There are two measures of agent performance that could be affected by the design. One is related to overhead: Does the design impose an unacceptable runtime overhead? The other is how well the agent performs in competition against other agents that have been implemented with different designs.

The Avalon framework does indeed impose some overhead when the agent starts up, since it must read configuration files, find and load code for components, and set up and configure the components. However, once the agent is running, there is essentially no overhead imposed by the framework. Event processing and evaluation is done by direct lookup, since event handlers and Evaluators are registered when components are loaded. We have run 6 MinneTAC agents on the same desktop machine (a 1 GHz Pentium), and all 6 agents are able to complete their daily decision procedures in less than 1 second.

MinneTAC did reasonably well in the first official TAC SCM competition, held in August 2003. It was eliminated in the semi-final rounds. It did well in high-demand games, but did a poor job of inventory management in low-demand games. Since then, new Sales, Production and Procurement components have been implemented and are being tested. The ease with which these new components could be implemented and configured into an agent is a testament to the design we describe here.

3.5.2 Usability

Mitch Kapor says that software design bridges the world of people and human purpose with the world of technology [9]. It’s easy to understand what that means when the artifact is a desktop application intended to be used directly by people. But in fact, the first user of a software design is the programmer who implements it. From the implementor’s standpoint, good design is easy to understand, easy to implement correctly, and easy to maintain.

MinneTAC is an autonomous agent. It exists in the game environment and has no user interface, other than the configuration files it reads and the logfiles it produces. The principal usability criterion is whether researchers can effectively work on the various decision problems independently, and whether they can extract the data they need to analyze performance and confirm or refute hypotheses. Inexperienced student programmers have been able to contribute significant functionality, and a wide variety of analyses have been carried out. An example of these analyses is in [12].

4 A Priori Game Analysis

Prior to the competition, we analyzed the game to determine its potential bottlenecks, where we define a *bottleneck* on day d as the factor which limits the production of PCs on day d . We identified three types of bottlenecks: (1) a *demand bottleneck*, which happens if the demand for profitable PCs is less than the agents' production capacities and the amount of available supplies, (2) a *production bottleneck*, which happens if the limiting factor is the agents' production capacities, and (3) a *supply bottleneck*, which happens when the limiting factor is the amount of available supplies.

To detect what bottlenecks might arise in a game, we start by estimating the maximum potentially profitable production of PCs on a day:

$$Production(d) = \min(Demand(d) \times BidFraction(d), ProductionCapacity(d), SuppliesAvailable(d)) \quad (1)$$

where

$$Demand(d) = \# RFQs(d) \times \overline{Quantity_RFQ}(d)^1 \quad (2)$$

$BidFraction(d)$ is the proportion of $Demand(d)$ which actually receives bids. If the reserve price specified in the RFQ is higher than the price of the components used to make the PC, an agent can make a profit by bidding at or below the reserve price, assuming it has production capacity and components. Since customers' reserve prices are chosen randomly in the interval of 75% to 125% of the maximum price of the components, approximately half of the RFQs will specify reserve prices that are higher than the cost of the components. Therefore, we can put a lower bound of 0.5 on $BidFraction$. Since components are often available at a discounted rate if ordered ahead of time, in some games $BidFraction$ may be as high as 1.0.

$ProductionCapacity(d)$ is the maximum number of PCs that can be produced on a day by the agents:

$$ProductionCapacity(d) = \frac{\# Cycles(d) \times \# Agents}{AvgCycles} = \frac{2000 \times 6}{5.5} = 2181 \quad (3)$$

where $AvgCycles$ is the average number of cycles to build a PC, which is computed assuming that each agent has sufficient supplies and that each of the 16 types of PCs is produced in equal quantities. Since the number of cycles needed to build a PC is between 4 and 7, the value of $ProductionCapacity(d)$ is in the range [1714, 3000].

$SuppliesAvailable(d)$ is the number of PCs that can be built from the supplies available in a day, assuming that suppliers are producing at maximal capacity and components in each category are produced with equal frequency. Every PC requires four components Thus

$$SuppliesAvailable(d) = MaxSupplies(d)/4 = \sum_{i=1}^8 Capacity_i(d)/4 \quad (4)$$

where $MaxSupplies(d)$ is the maximum number of supplies produced in a day. A supplier i has a production capacity $Capacity_i$, which is determined on each day d by a mean reverting random walk with a lower bound:

$$Capacity_i(d) = \max(0, Capacity_i(d-1) + random(-0.05, 0.05) \times NominalCapacity + 0.01 \times NominalCapacity - Capacity_i(d-1))$$

where $NominalCapacity$ is the nominal capacity, which is specified as 500 components per day. $Capacity_i(0) = NominalCapacity$ is used to compute $Capacity_i(1)$, the supplier's production capacity on the first day of the game.

We combine the results of Equations 1, 2, 3, and 4 to predict the maximal number of potentially profitable PCs production in a day d :

$$Production(d) = \min \left(\# RFQs(d) \times \overline{Quantity_RFQ}(d) \times BidFraction, \frac{\# Cycles(d)}{AvgCycles} \times \# Agents, \sum_{i=1}^8 Capacity_i(d)/4 \right) \quad (5)$$

¹ the notation \bar{x} denotes the sample mean of the variable x .

In a typical game, the initial average number of customer RFQs per day is 200. For simplicity, we will assume it is always 200, and that suppliers' daily capacity is always the nominal capacity of 500 components per day. We assume that $BidFraction = 0.5$; that is, we assume that agents are not able to obtain supplies at a discounted rate. We also assume $\overline{Quantity_RFQ} = 10.5$, since the average quantity specified in each RFQ is chosen randomly from a uniform distribution over the interval $[1, 20]$. By substituting these values into Equation 1 we obtain: $Production(d) = \min(200 \times 10.5 \times 0.5, 2000 \times 6/5.5, \sum_{i=1}^8 500/4) = \min(1050, 2181, 1000) = 1000$. This result shows that the most likely bottleneck for a typical game is supply availability. In fact, the availability of supplies is the most probable bottleneck as long as the number of RFQs per day is greater than 190. (If $\# RFQs = 190$, then $E[Demand(d)] \times BidFraction(d) = 190 \times 10.5 \times 0.5 = 997.5 < 1000$.)

Since the initial average number of customer RFQs is chosen randomly from a uniform distribution over the range $[80, 320]$, approximately 45.8% of games will start off with a demand bottleneck. A greater number of games might enter a demand bottleneck after some period of time due to fluctuations in the average number of customer RFQs. In the above calculations we assumed that $BidFraction = 0.5$. If agents get supplies at a discounted price, as competition data indicate, then we expect that $BidFraction$ will be greater than 0.5. The availability of supplies is then even more likely to be a bottleneck. The above calculations also suggest that agents' production capacities are not likely to be a bottleneck in any game in which there are at least three functioning agents. However, a single agent could be limited by its production capacity if it acquires substantially more supplies than its opponents.

5 Performance Analysis of MinneTAC Sales Strategies

Building to order would be a good strategy in the case of a demand bottleneck. However, since the most likely bottleneck is a supply bottleneck, we decided to use a supply-driven strategy.

Ideally, an agent's strategy should be flexible and adjust dynamically in each game without *a priori* assumptions. Because of the first day discount, an agent is better off making an uninformed *a priori* estimate of customer demand and ordering most of the components it anticipates using on the first day. The drawback to this approach is that if an agent acquires too many components, it may be forced to sell them at loss, since the components have no value at the end of the game.

5.1 MinneTAC sales strategies

We designed, implemented, and compared two variants of a supply-driven sales strategy, that we call *MaxEProfit* and *DemandDriven*. To compare these strategies we have created two variants of the MinneTAC agent, called Tabaluka and Eini. Tabaluka uses *MaxEProfit*, MinneTAC and Eini use *DemandDriven*.

Each day the agent determines an offer price for each RFQ. Offers are made only from the uncommitted finished goods inventory and are sorted by decreasing profit margin, where $ProfitMargin = (price - cost)/price$. The strategies differ in the way they set prices and they estimate the probability of receiving an order. For each offer made the agent reserves a fraction of the computers offered according to the probability of receiving an order, $P(order)$.

5.1.1 MaxEProfit

MaxEProfit determines an offer price that maximizes the expected profit margin from an order:

$$E[ProfitMargin] = ProfitMargin \times P(order)$$

with the constraint that $price \geq TargetAveragePrice$. $ProfitMargin$ is calculated on the agent's moving average cost of the components. $TargetAveragePrice$ is an internal parameter that reflects the current market prices. The parameter is adjusted every 5-days based on the orders received, and every 20-days based on the market report, time left in the game, production rate, uncommitted finished goods inventory level, and customer demand. This parameter helps to ensure that the agent is not left with winning only unprofitable RFQs.

$P(order)$ is the estimated probability of receiving a customer order for the price offered. This probability is influenced by the reserve price, quantity, lead time, penalty, and product type, in addition to the price specified in the offer. *MaxEProfit* models this probability as a 6-dimensional *OrderProb* matrix with the following dimensions: price,

quantity, lead_time, reserve_price, penalty, and product_type. Each entry in *OrderProb* contains the probability that a customer will accept an offer given the values of the parameters. Initially the values are all set to 1, making the agent optimistic. The values are updated during the game whenever an offer is accepted or rejected, with a predefined learning rate.

5.1.2 DemandDriven

DemandDriven sets a desired target probability, *TargetProb*, of receiving an order, and determines an offer price accordingly. *TargetProb* is computed from the reverse cumulative density function (CDF) that models the order probability. The objective is to make offers to sell out the inventory by the end of the game. *TargetProb* (see Equation 6) is calculated every 5-days based on the currently observed market conditions (customer demand, and time left in the game) and on internal parameters (production rate and uncommitted finished goods inventory for a specific product).

$$TargetProb = \min(1, \frac{InventoryPCs + Production \times DaysLeft}{EstimatedDemand}) \quad (6)$$

where *InventoryPCs* is the number of PCs available in the inventory for a particular product. *Production* is the number of units built each day, *DaysLeft* is the number of days left in the game, and *EstimatedDemand* is an internal parameter that forecasts future demand.

DemandDriven models the probability of a customer order as a 5-dimensional *OrderProb* matrix having the following dimensions: price, customer_demand, lead_time, reserve_price, and product_type. The values of customer_demand are discretized into 3 ranges: low, medium, and high; lead_time is discretized into short and long. The matrix is pre-populated with values obtained from analysis of several past games. DemandDriven assumes that only a shift of the whole order probability curve could occur during the game, so, instead of updating the values of *OrderProb* as done by MaxEProfit, every five days it shifts the values toward higher or lower prices depending on the difference between the acceptance rate of its offers and *TargetProb*.

Most games can be classified as either high or low customer demand [7]. Table 1 compares the results of the two strategies over a series of high-demand² and low-demand games³ all played on tac5.sics.se:8080. We can see that MaxEProfit performs better in high-demand games, but is generally worse in low-demand games.

Strategy	High			Low		
	Values (in \$M)					
	Min	Avg	Max	Min	Avg	Max
MaxEProfit (Tabaluka)	-12.02	12.30	35.99	-66.90	-44.44	-7.36
DemandDriven (Eini)	-23.65	8.70	30.89	-57.15	-34.49	30.89

Table 1: Performance comparison of MaxEProfit and DemandDriven strategies in high and low-demand games.

Game Number	Agents and their Result (in \$M)						Customer Demand in RFQs		
	1	2	3	4	5	6	#RFQ	#RFQ/day	σ
2214	team2	RedSox	MinneTAC	arnoch	RedAgent	Eini	21778	99.44	51.8
	-10.43	-18.3	-31.06	-34.87	-38.08	-39.83			
2218	Tabaluka	RedAgent	arnoch	MinneTAC	team2	Eini	65626	299.66	42.14
	31.23	30.69	23.24	20.8	8.86	7.89			

Table 2: Summary of the games examined. #RFQ is the total number of RFQs during the game, #RFQ/day is the mean number of RFQs/day, and σ is the standard deviation. Customer RFQs are issued over 219 days in a game. Eini and Tabaluka are variants of MinneTAC. Eini and MinneTAC use DemandDriven, Tabaluka uses MaxEProfit.

²2385,2393,2396,2401,2407,2409,2410,2411,2412,2416, 2419,2420,2421,2594,2597,2598,2603,2607,2612,2613

³2383,2387,2390,2392,2394,2399,2415,2417,2423,2425, 2426,2492,2593,2595,2599,2600,2604,2610,2614,2640

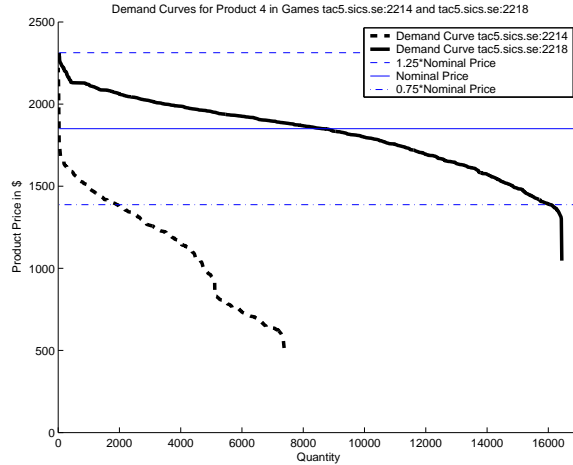


Figure 8: Games 2214 and 2218 – Aggregate demand curves for product 4 over the games.

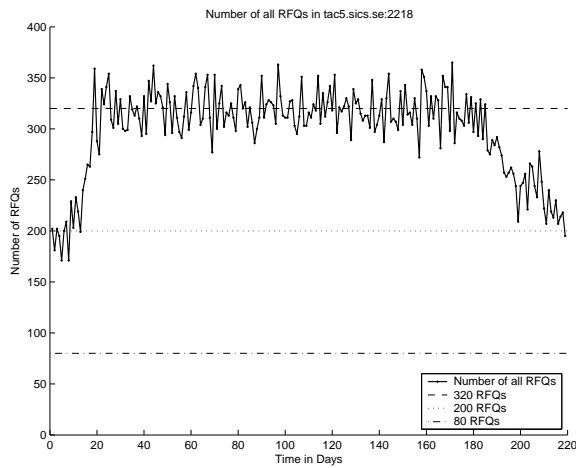


Figure 9: Games 2218 – Total number of RFQs. This is a high-demand game.

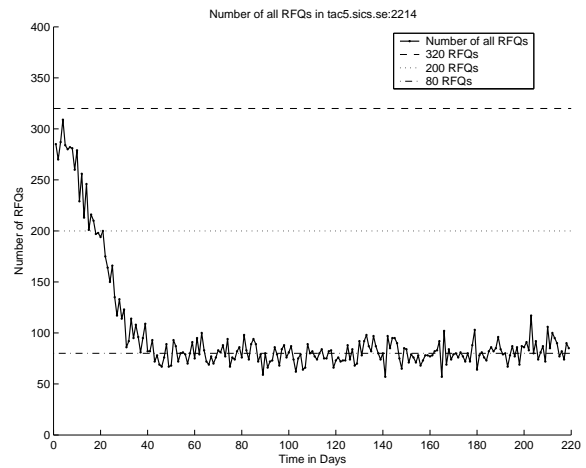


Figure 10: Game 2214 – Total number of RFQs. This is a low-demand game.

We will now analyze our two strategies in a high-demand and a low-demand game (see Table 2). We focus our comparison on product 4, whose nominal price (which is the sum of the base cost of the four individual components) is \$1850. Similar results can be shown for the other products.

In Figure 8, we show the aggregate demand curve for product 4 over the two games by price. The aggregate demand curve for a high-demand game is very different from the curve for a low-demand game. In the high-demand game 9 there is demand for computers at prices above the nominal price, but in low-demand games, as the game in Figure 10, the bulk of the demand is even below the minimum reserve price, which equals 75% of the nominal price.

In Figure 11, we show the probability of order curves for all products in the game 2218, high-demand. In the adjacent Figure 12 we show the same curves, but for the game 2214, low-demand. We see that the probability of order curve is quite different in these two situations. Clearly, the probability of order curve at low-demand is not just a shifted version of the high-demand curve.

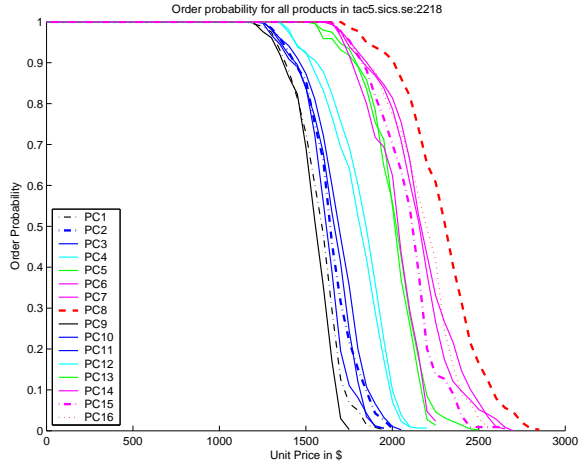


Figure 11: Games 2218 – Probability of order curve (CDF). This is a high-demand game.

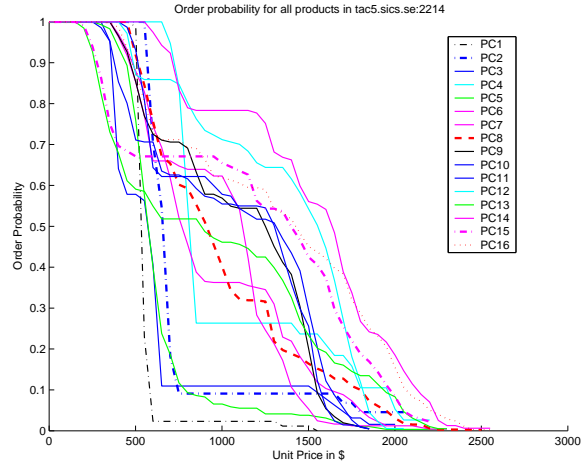


Figure 12: Game 2214 – Probability of order curve (CDF). This is a low-demand game.

5.2 Performance in high-demand games

We show now how MaxEProfit performs in a high-demand game. In Figure 13 we show the market reports every twenty days and compare them with the agent’s predictions. Since predictions are updated every 5 days, we show the agent predictions and the real prices over the same periods. In addition we show the average offer prices (circles) made and the average order prices (crosses) received.

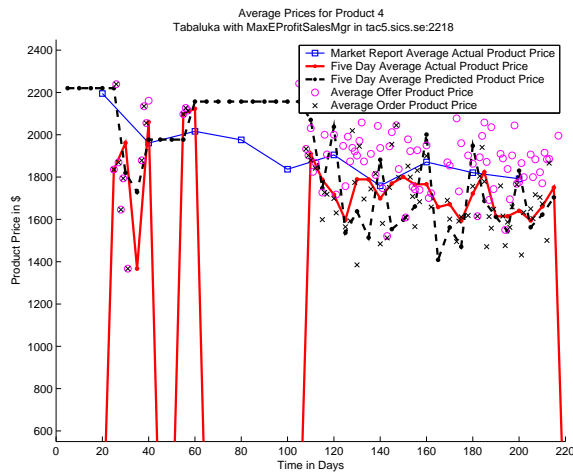


Figure 13: Game 2218 – Timeline for market report: Predicted vs actual values of Tabaluka (MaxEProfit) for product 4.

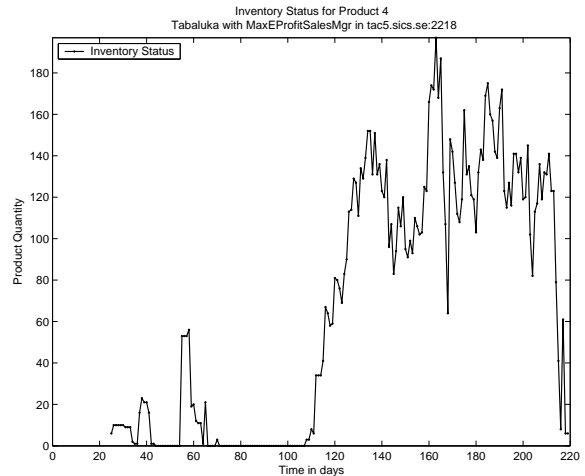


Figure 14: Game 2218 – Timeline for finished goods inventory of Tabaluka (MaxEProfit) for product 4.

The inventory of finished goods (see Figure 14) was mostly empty until halfway through the game, and when the inventory is empty the sales strategy doesn’t learn and makes no offers. This situation can be seen as a straight line in Figure 13 for the predicted product price. We observe that predicted product prices match well actual product prices, even though the prediction often over- and undershoots the real values.

Towards the end of the game this sales strategy tries to sell out the uncommitted finished goods inventory if the finished goods inventory level is higher than what the agent thinks it will be able to sell. We can see the relationship

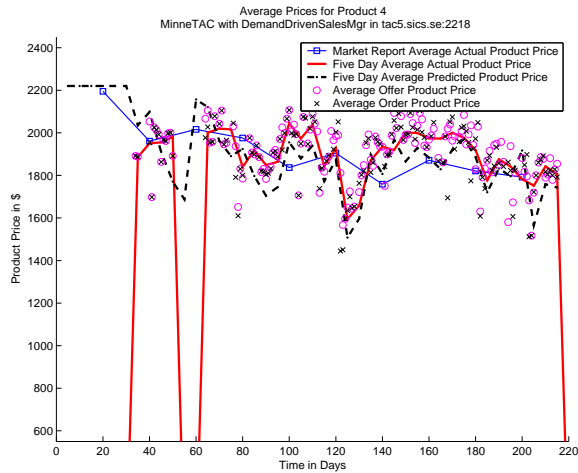


Figure 15: Game 2218 – Timeline for market report: Predicted vs actual values of MinneTAC (DemandDriven) for product 4.

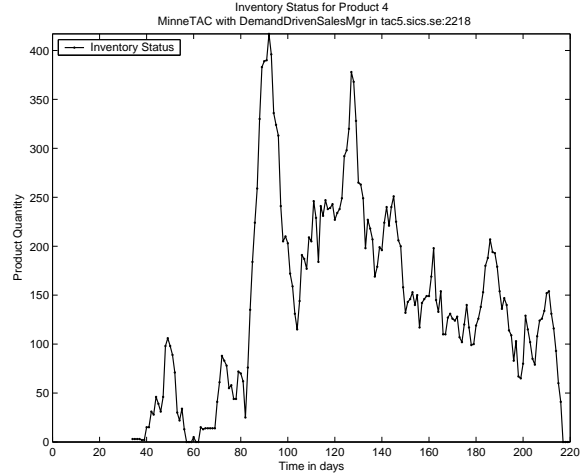


Figure 16: Game 2218 – Timeline for finished goods inventory of MinneTAC (DemandDriven) for product 4.

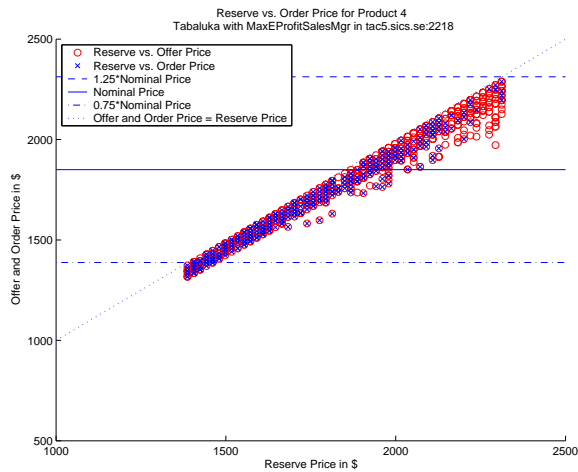


Figure 17: Game 2218 – Offer and order prices of Tabaluka (MaxEProfit) vs reserve price for product 4.

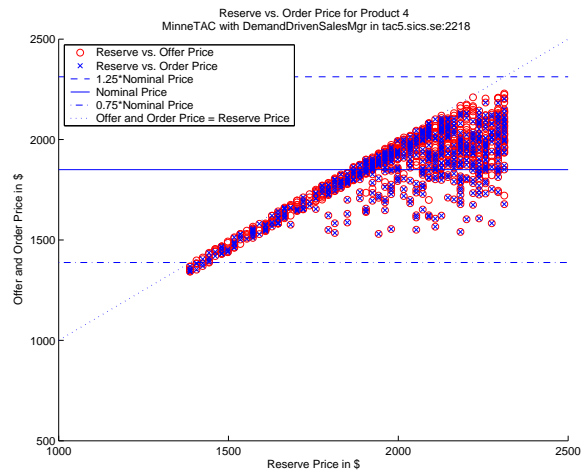


Figure 18: Game 2218 – Offer and order prices of MinneTAC (DemandDriven) vs reserve price for product 4.

between offer/order prices and reserve prices in Figure 17. A circle with a cross inside symbolizes an accepted order. We observe that in this case the reserve and order prices are close. This is not the case in every game.

We now analyze the performance of MinneTAC, which uses the DemandDriven strategy, in the same game. Figure 15 shows the same information as Figure 13, but for DemandDriven. We can see the relationship between offer/order prices and reserve prices of MinneTAC in Figure 18. We observe that in this case the reserve and order prices are not as close, as when using MaxEProfit in the same game (see Figure 17). The reason is that DemandDriven tries to clear the inventory by the end of the game, and it is willing to make offers far below the reserve price. DemandDriven is often too aggressive in selling goods because it assumes that a constant supply of raw material is being supplied until the end of the game.

5.3 Performance in low-demand games

We will now analyze the performance of the DemandDriven strategy in low-demand games. MaxEProfit is not discussed here, since the analysis is similar. The first day the agent orders large amounts of components. The sales strategy starts to make offers only when there are finished goods in the inventory. Whenever finished goods are almost sold out, MinneTAC purchases new components to retain a certain level of raw inventory. This is problematic in very low-demand games, since the agent ends up with too many components and finished goods in the inventory.

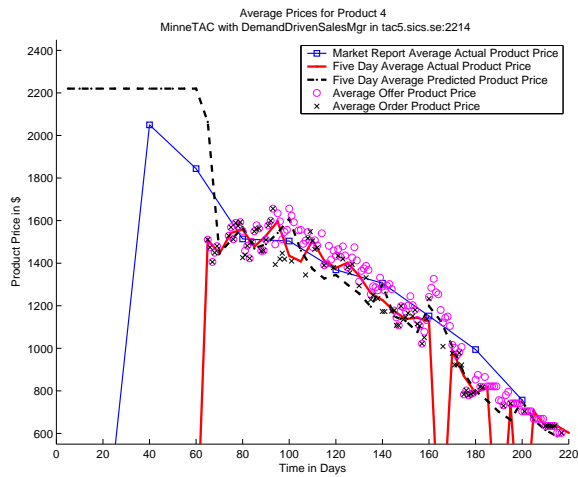


Figure 19: Game 2214 – Timeline for market report: Predicted vs actual values of DemandDriven for product 4.

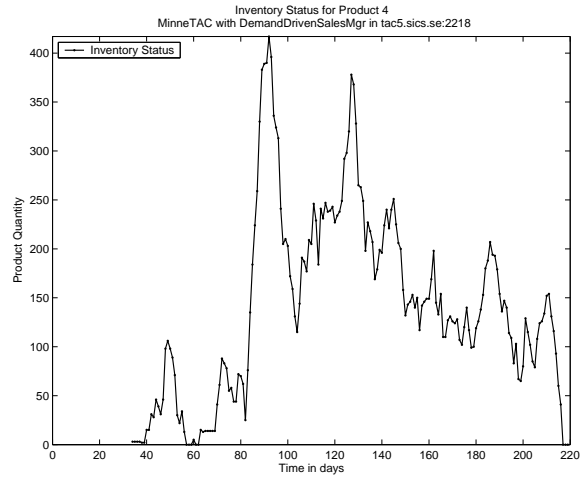


Figure 20: Game 2218 – Timeline for finished goods inventory of MinneTAC (DemandDriven) for product 4.

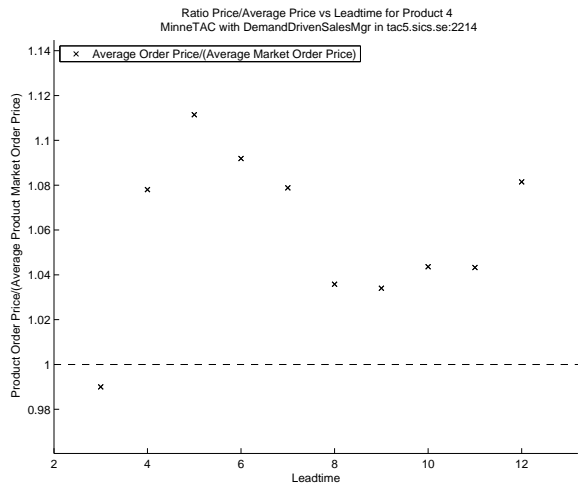


Figure 21: Game 2214 – Lead time vs average MinneTAC order price over average market order price for product 4.

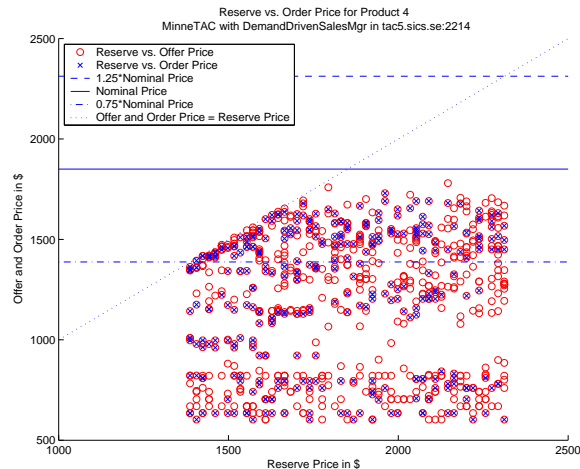


Figure 22: Game 2214 – Offer and order prices of MinneTAC (DemandDriven) vs reserve price for product 4.

In Figure 19 we can see the agent’s predictions versus the market reports, and the orders and offers made through the game. We can observe how well the prices offered tracked the real market price. Offer and order prices decrease with respect to the reserve prices as the game progresses (see Figure 19). In a low-demand situation like this, the competition to sell out products is very high and therefore the prices lower so much. In high-demand games the prices are usually between 75% and 125% of the nominal product price (see Figures 17 and 18), while the bulk of the orders

Agent	Total Values (in \$M) of					DM (days)
	RFQs	Timely Offers	Orders	Discount	Final Result	
Tabaluka	142.93	57.90	118.48	59.24	31.23	73.57
RedAgent	132.03	0.00	14.80	57.40	30.69	91.94
arnoch	79.00	7.97	42.24	21.12	23.24	92.40
MinneTAC	142.93	28.31	95.46	47.73	20.80	102.39
team2	2.00	0.00	2.00	1.00	8.86	51.35
Eini	142.93	21.18	72.27	36.13	7.89	119.63

Table 3: Start-effect for Game 2218: RFQ is the value of the components requested the first day, Timely Offers is the value of the components offered at the requested time, Orders is the value of the components ordered, and Discount is the discount (50%) obtained. DM is the delay measure in days.

in this low-demand game is far below 75% of the nominal product price (see Figure 22).

6 Analysis of Start-Effect

We will now analyze the start-effect in the game and show that the total volume an agent orders on the first day and the timeliness of the offers that it accepts have a strong impact on that agent’s final score. We developed a start-effect measure called *delay measure* (DM). DM (see Equation 7) is the delay, in days, in delivering the components weighted by the component total value.

$$DM = \frac{\sum_{i=1}^{\#RFQs} Quantity(RFQ_i) \times BasePrice(RFQ_i) \times DueDate(Offer_i)}{\sum_{i=1}^{\#RFQs} Quantity(RFQ_i) \times BasePrice(RFQ_i)} \quad (7)$$

where *BasePrice* is the component’s base price and *DueDate* is the due date of the offer. Results for game 2218 are shown in Table 3. The measure works very well for high-demand games, but not as well for low-demand games. We can conclude that a low value of DM leads to a better final score. Tabaluka had the lowest DM and ended up first, Eini had the highest DM and came in last. A low DM is an effective indicator only when a high volume of goods is ordered on the first day. In this game team2’s low DM is not a good indicator of its final result because of its low order volume on the first day.

In addition to looking at single games we looked at many high-demand and low-demand games played at the 2003 International Conference for Electronic Commerce (Pittsburgh, October 2003). The games make a good test set because the configurations for each agent didn’t change and the agents were robust. To determine if a game is high-demand or low-demand, we looked at $ratio = \frac{\sum \#ComputersOrdered}{ActivePlayers}$ and selected the 20 games with the highest⁴ and lowest⁵ ratios. Then we calculated the correlation coefficients between the bank status at the end of the game, the volume of first day orders, and DM.

In high-demand games we calculated a correlation coefficient of 0.5381 between bank status and the total amount ordered on the first day, and a correlation coefficient of -.03456 between bank status and DM. This shows that in high-demand games there is a strong relationship between the amount of parts an agent orders on the first day and its final score. There is also a strong relationship between DM and the final score, perhaps because agents which received better offers were more likely to accept them. This indicates that the order in which suppliers process RFQs has a strong impact on the outcome of high-demand games.

In low-demand games the correlation coefficient between bank status and the total amount ordered on the first day was -0.3242 and the correlation coefficient between bank status and DM was 0.3904. This indicates that agents who did not order a lot of parts (perhaps because they received poor offers on the first day) did better in low-demand games, because they were less likely to be trapped with inventory they could not sell at a profit.

⁴ 1641,1646,1650,1651,1652,1660,1661,1662,1663,1665, 1666,1667,1670,1673,1674,1682,1683,1685,1686,1690

⁵ 1640,1642,1643,1644,1649,1653,1654,1657,1658,1659, 1668,1669,1671,1672,1676,1679,1680,1681,1687,1688

7 Related Work

Most agent design efforts have focused on either the autonomous behavior aspects of agency, or on interaction among agents. Shoham's Agent-Oriented Programming [18] examines a cognitive and societal view of computation. Bradshaw's KAOs agents [3] are BDI agents in a CORBA environment. Agents have *capabilities* based on existing document management applications. Norman *et al.* [16] describe agent societies that model organizational structures and automate business processes. These ADEPT agents negotiate over service agreements that can involve many parties and many dimensions. JADE [15] is an agent framework that has been used to build trading agents, and could have been used for MinneTAC. However, its primary emphasis is on building multi-agent systems that comply with FIPA specifications for inter-agent communications, and with flexible deployment in a network environment. This is not a requirement for the TAC SCM domain. The MinneTAC design is *compositional* in the sense of Brazier *et al.* [4], but not hierarchically so. The DESIRE method from Brazier *et al.* does not seem applicable to the MinneTAC situation, since we are dealing with a single agent in an existing environment, and the blackboard approach used in MinneTAC is not easily modeled with DESIRE. RETSINA [19] suggests both a multi-agent architecture with a variety of agent roles, and an architecture for individual agents that provides communications, planning, scheduling, and execution monitoring. This architecture could probably be adapted to the TAC SCM domain, but its planning and communication capabilities would not be especially useful. Vetsikas and Selman [21] show a method for studying design tradeoffs in a trading agent. This approach could be likely be used effectively in MinneTAC.

Predicting prices is an important part of the decision process of agents. Our strategies have been inspired by the work of Kephart *et al.* [11], who explored several dynamic pricing algorithms for information goods, where shopbots look for the best price, and pricebots try to adapt their prices to attract business. Wellman [22] analyzed and developed metrics for price prediction algorithms in the TAC Classic game, similar to what we have done for TAC SCM.

The teams which played in TAC SCM have used different strategies for price and prediction of probability of order. In [13] three key issues of TAC SCM are identified: (1) *uncertainty* about the environment and future events, (2) *dynamism* - decisions can ripple through the supply chain over time, and (3) *strategic* - attempting to influence competitors through the shared environment.

Nearly all successful agents in last year's competition used some way of modeling the probability of receiving an order. For instance, Botticelli [2] uses a linear cumulative density function (CDF) to determine the relationship between offer price and customer order probability, while we used a reverse CDF.

TacTex [17] uses the lowest and highest offer price, which is provided for each product every day by the game server, to determine the probability of a customer order. From these values it calculates the lowest price, the average low price, the midpoint between the average low and average high price, the average high price, and the highest price for each computer type each day, and estimates the probability of an order by linear interpolation of the range where its offer price falls. Their estimates depend only on the type of computer requested and the reserve price, whereas we use more parameters (6 for the MaxEProfit strategy and 5 for the DemandDriven strategy).

Instead of calculating the probability of order, RedAgent [10], the winner of last year TAC SCM used an internal marketplace structure with competing bidder agents to set offer prices. PackaTAC [6] used what they called a *following* strategy, in which instead of trying to undercut other agents, PackaTAC lets them set the price and tries to follow. The Jackaroo team [23] applied a game theoretic approach to compute a market clearing price.

Since we estimated the bottleneck was going to be in the supply and not in the production, we did not worry about optimizing the production of our agent. Other teams [2, 17] instead have focused on optimizing production to avoid late penalty costs. The poor performance of MinneTAC was not caused by missing shipments but mostly by ordering too many supplies the first day of the game. This proved to be specially critical in low-demand games. Because of the bimodal nature of the games, which were either low or high-demand [7], a less aggressive buying strategy early in the game would have given the agent time to assess the market situation more accurately. In addition, MinneTAC was too eager to sell computers, and often sold them too early in the game rather than waiting for better market conditions.

8 Conclusions and Future Work

Experimental work with multi-agent systems requires implementations. Often, the design qualities that best support experimental work are different from those normally considered "ideal" in industry. In complex economic scenarios

such as the Supply Chain Trading Agent Competition, the desired design qualities include clean separation of infrastructure from decision processes, ease of implementation of multiple decision processes, clean separation of different decision processes from each other, and controllable generation of experimental data. In a competition environment, the ability to compose multiple agents with different combinations of decision process implementations makes it possible to test hypotheses about the effectiveness of competing decision models.

We show one way to construct such an agent, using a readily-available component framework. The framework provides the ability to compose agent systems from sets of individual components based on simple configuration files. We also show that two basic mechanisms, event distribution with the Observer pattern and our Evaluator-Evaluation scheme, permit an appropriate level of component interaction without introducing unnecessary coupling among components. There are many possible extensions to the basic design we show here. One that we are currently pursuing is to add an “executive” component that would allocate “resources” to competing implementations of basic decision processes within a single agent. This would allow a high degree of adaptability in the game environment, where the level of demand can fluctuate greatly, and where the actions of other agents can have a significant impact on the markets.

We analyzed the factors which can limit the performance of an agent in TAC SCM. The limit to profitability is usually either caused by limited customer demand or limited supplier capacity. MinneTAC orders parts based on the assumption of a supplier capacity bottleneck (high-demand game). We have explored two different sales strategies to compensate in the case of a customer demand bottleneck (low-demand game). Both MinneTAC’s sales strategies were competitive in high-demand games, but often sold larger quantities at lower margins compared to other agents. Even though the DemandDriven strategy performs better than the MaxEProfit strategy in low-demand games it is still unable to compensate for the large number of parts ordered on the first day.

We have shown that the random order in which agent’s RFQs are selected by suppliers and the huge discount on components ordered the first day affect the outcome of the game so much that for high-demand games the winner of the game can be predicted from the first day replies of suppliers. We have developed a measure of the delay in obtaining supplies, and shown that the a low value of delay correlates with good performance in high-demand games. However, in low-demand games a low delay correlates with bad performance.

The TAC SCM rule-set will undergo significant changes for the 2004 competition, which will affect some parts of our analysis. Customer demand will be evened out so that there will no longer be a clear distinction between high-demand and low-demand games. Changes to supplier pricing rules will reduce the start-effect by making it more difficult to acquire large quantities of cheap materials on the first day. There are a number of areas where the strategy of our agent could be improved, and we are in the process of exploring some of them.

9 Acknowledgement

We would like to express our gratitude to the many teams who participated and to the people who organized 2003 TAC SCM, especially R. Arunachalam, J. Eriksson, N. Finne, S. Janson, and N. Sadeh. Partial support for this research is gratefully acknowledged from the National Science Foundation under award NSF/IIS-0084202.

References

- [1] Raghu Arunachalam, Joakim Eriksson, Niclas Finne, Sverker Janson, and Norman Sadeh. The TAC supply chain management game. Technical Report Draft Version 0.62, Swedish Institute of Computer Science, Sweden, 2003.
- [2] Michael Benisch, Amy Greenwald, Ioanna Grypari, Roger Lederman, Victor Naroditskiy, and Michael Tschantz. Botticelli: A supply chain management agent designed to optimize under uncertainty. *ACM Trans. on Computational Logic*, 4(3):29–37, 2004.
- [3] Jeffrey M. Bradshaw, Stewart Duffield, Pete Benoit, and John D. Wolley. KAoS: Toward an industrial-strength open agent architecture. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 375–418. AAAI Press, 1997.
- [4] Frances M. T. Brazier, Catholign M. Jonker, and Jan Treur. Principles of component-based design of intelligent agents. *Data and Knowledge Engineering*, 41(1):1–28, April 2002.

- [5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: a System of Patterns*. Wiley, 1996.
- [6] Erik Dahlgren and Peter Wurman. PackaTAC: A conservative trading agent. *SIGecom Exchanges*, 4(3):33–40, 2004.
- [7] J. Estelle, Y. Vorobeychik, M.P. Wellman, S. Singh, C. Kiekintveld, and V. Soni. Strategic interactions in a supply chain game. Technical report, University of Michigan, USA, 2003.
- [8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.
- [9] Mitchell Kapor. A software design manifesto. In Terry Winograd, editor, *Bringing Design to Software*, pages 1–9. ACM Press, 1996.
- [10] Philipp W. Keller, Felix-Olivier Duguay, and Doina Precup. Redagent - winner of the TAC SCM 2003. *SIGecom Exchanges*, 4(3):1–8, 2004.
- [11] Jeffrey O. Kephart, James E. Hanson, and Amy R. Greenwald. Dynamic pricing by software agents. *Computer Networks*, 32(6):731–752, 2000.
- [12] Wolfgang Ketter, Elena Kryzhnyaya, Steven Damer, Colin McMillen, Amrudin Agovic, John Collins, and Maria Gini. MinneTAC sales strategies for supply chain TAC. In *Proc. of the Third Int’l Conf. on Autonomous Agents and Multi-Agent Systems*, page submitted, New York, July 2004.
- [13] C. Kiekintveld, M. P. Wellman, S. Sing, J. Estelle, Y. Vorobeychik, V. Soni, and M. Rudary. Distributed feedback control for decision making on supply chains. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling*, 2004.
- [14] B. Loritsch. Developing with Apache Avalon. Apache Software Foundation, 2001.
- [15] P. Moraitis, E. Petraki, and N. Spanoudakis. Engineering JADE agents with the Gaia methodology. In R. Kowalszyk, J. Miller, H. Tianfi eld, and R. Unland, editors, *Agent Technologies, Infrastructures, Tools, and Applications for e-Services*, volume 2592 of *Lecture Notes in Computer Science*, pages 77–91. Springer-Verlag, 2003.
- [16] Timothy J. Norman, Nicholas R. Jennings, Peyman Faratin, and E. H. Mamdani. Designing and implementing a multi-agent architecture for business process management. In M. J. Wooldridge, J. P. Müller, and N. R. Jennings, editors, *Intelligent Agents III*, volume 1193 of *Lecture Notes in Artificial Intelligence*, pages 261–275. Springer-Verlag, Berlin, 1997.
- [17] David Pardoe and Peter Stone. TacTex-03: A supply chain management agent. *SIGecom Exchanges*, 4(3):19–28, 2004.
- [18] Yoav Shoham. An overview of agent-oriented programming. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 271–290. AAAI Press, 1997.
- [19] Katia Sycara and Ananddeep S. Pannu. The RETSINA multiagent system: towards integrating planning, execution, and information gathering. In *Proc. of the Second Int’l Conf. on Autonomous Agents*, pages 350–351, 1998.
- [20] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, 1998.
- [21] Joannis A. Vetsikas and Bart Selman. A principled study of the design tradeoffs for autonomous trading agents. In *Proc. of the Second Int’l Conf. on Autonomous Agents and Multi-Agent Systems*, 2003.
- [22] Michael P. Wellman, Daniel M. Reeves, Kevin M. Lochner, and Yevgeniy Vorobeychik. Price prediction in a trading agent competition. *Journal of Artificial Intelligence Research*, 2003.
- [23] Dongmo Zhang, Kanghua Zhao, Chia-Ming Liang, Gonelur Begum Huq, and Tze-Haw Huang. Strategic trading agents via market modeling. *SIGecom Exchanges*, 4(3):46–55, 2004.