Flexible decision control in an autonomous trading agent

John Collins [a,*], Wolfgang Ketter [b], Maria Gini [a,1]

[a] *Dept. of Computer Science and Engineering, University of Minnesota, 4-192 EE/CS Bldg., 200 Union St SE, Minneapolis, MN 55455, USA.*

[b] *Department of Decision and Information Sciences, Rotterdam Sch. of Mgmt., Erasmus University, 3000 DR Rotterdam, NL*

Abstract

Modern electronic commerce creates significant challenges for decision-makers. The Trading Agent Competition for Supply Chain Management (TAC SCM) is an annual competition among fully-autonomous trading agents designed by teams around the world. Agents attempt to maximize profits in a supply-chain scenario that requires them to coordinate procurement, production, and sales activities in competitive markets. An agent for TAC SCM is a complex piece of software that must operate in an economic environment where information is only partially visible. We report on results of an informal survey of agent design approaches among the competitors in TAC SCM, and then we describe and evaluate the design of our MinneTAC trading agent. We focus on the use of *evaluators* – configurable, composable modules for data analysis, modeling, and prediction that are chained together at runtime to support agent decision-making. Through a set of examples, we show how this structure supports sales and procurement decisions, and how those decision process can be modified in useful ways by changing evaluator configurations.

Key words: Trading Agent Competition, multi-agent systems, software architecture, supply chain management.

* Corresponding author.

Email addresses: jcollins@cs.umn.edu (John Collins), wketter@rsm.nl (Wolfgang Ketter), gini@cs.umn.edu (Maria Gini).

[1] Partial support from the National Science Foundation under award NSF/IIS-0414466.

1 Introduction

Electronic commerce is a compelling application area for autonomous agents. On one side, 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. On the other side, an economically-motivated autonomous agent must not only sense and act in its environment; it must also compete. Depending on the details of the market environment, rational behavior may involve strategic elements in addition to simple utility-maximization.

Organized competitions can be an effective way to drive research and understanding in complex domains, free of the complexities and risk of operating in open, real-world economic environments. Artificial economic environments typically abstract certain interesting features of the real world, such as markets and competitors, demand-based prices and cost of capital, and omit others, such as personalities, taxes, and seasonal demand.

Our primary interest in this paper is to examine some of the software design tradeoffs in building an autonomous agent for the Supply-Chain Management Trading Agent Competition [13] (TAC SCM), and to describe in some detail the design of the MinneTAC trading agent, which has competed effectively in TAC SCM for several years.

We describe how 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 also evaluate the success of our design in terms of the performance and competitiveness of the agents that have been implemented with it, the usability of the design and its core abstractions from the standpoint of researchers who must work within its confines, and against a set of standard design-quality metrics.

In Section 2, we review the TAC SCM game scenario, focusing on the decision challenges presented by that scenario. We discuss the results of a survey we have conducted into the design philosophy and features of other agents designed for the same competition scenario. Section 4 provides a high-level overview of the design of MinneTAC, focusing on the core framework of the agent and its use of events and evaluators to structure the agent's decision processes. Section 5 provides multiple examples of evaluator-supported decision processes, and shows how they may be reconfigured both manually through configuration files, and automatically by the agent itself using selectors. Finally, Section 6 presents a brief evaluation of the architecture.

2 Overview of the TAC SCM game

In a TAC SCM game [13], each of the competing agents plays the part of a manufacturer of personal computers. Agents compete with each other in a procurement market for computer components, and in a sales market for customers. A typical game runs for 220 simulated days over about an hour of real time. Each agent starts with no inventory and an empty bank account, and must borrow (and pay interest) to build up inventory of computer components before it can begin assembling and shipping computers. The agent with the largest bank account at the end of the game is the winner.

2.1 Game scenario

Customers express demand each day by issuing a set of Requests for Quotes (RFQs) for finished computers. Each RFQ specifies the type of computer, a quantity, a due date, a reserve price, and a penalty for late delivery. Each agent may choose to bid on some or all of the day's RFQs. For each RFQ, the bid with the lowest price will be accepted, as long as that price is at or below the customer's reserve price. Once a bid is accepted, the agent is obligated to ship the requested products by the due date, or it must pay the stated penalty for each day the shipment is late. Agents do not see the bids of other agents, but aggregate market statistics are supplied to the agents periodically. Customer demand varies through the course of the game by a random walk.

Agents assemble computers from parts, which must be purchased from suppliers. When agents wish to procure parts, they issue RFQs to individual suppliers, and suppliers respond with bids that specify price and availability. If the agent decides to accept a supplier's offer, then the supplier will ship the ordered parts on or after the due date (supplier capacity is variable). Supplier prices are based on current uncommitted capacity.

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

2.2 Agent decisions

An agent for the TAC SCM scenario must make the following four basic decisions during each simulated "day" in a competition:

(1) decide what parts to purchase, from whom, and when to have them delivered (Procurement).
(2) schedule its manufacturing facility (Production).
(3) decide which customer RFQs to respond to, and set bid prices (Sales).
(4) ship completed orders to customers (Shipping).

These decisions are supported by models of the sales and procurement markets, and by models of the agent's own production facility and inventory situation.

The details of these models and decision processes are the primary subjects of research for participants in TAC SCM. In particular, the Sales and Procurement markets are highly variable, and many important factors, such as current capacity, outstanding commitments of suppliers, sales volumes and price distribution in the customer market, are not visible to the agents. In addition, the number of competing agents in the competition scenario is relatively small (just 6 in a single simulation). This means that agents have oligopoly power – the actions of individual agents can have a significant effect on the markets, and agents are motivated to engage in strategic manipulation of the markets to the extent allowed by the rules of the competition.

Beyond the challenges presented by the TAC SCM problem domain, our research needs present several additional issues. The most important is that 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. A good 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. In addition, 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 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 [27]. To understand the effect of settings the various parameters used in an agent, we need to be able to configure agents with different combinations of decision processes and their underlying models and parameters.

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.

3 TAC SCM Agent Design

We present here the results of an informal survey of research and development practices and related design issues for autonomous trading agents among participants in the Trading Agent Competition for Supply-Chain Management. The goal of the questionnaire was to understand the commonality and variability of design principles for agents competing in TAC SCM. We report our findings from a questionnaire that we sent to the TAC SCM community via the TAC SCM discussion email list in May 2007. The questionnaire was closed by September 2007 and was completed by many of the best teams in previous tournaments. We also supplement the practitioners' gained wisdom with relevant academic and industry papers. The questions used in the survey are given in an appendix to this paper. Table 1 lists the teams who responded to the questionnaire, along with their institutions and the survey respondents (in most cases, a student and a faculty advisor).

After a review of the completed questionnaires, we categorized the results according to our understanding of the research agendas of the teams, and by the specific architectural emphases the teams identified in their agent designs that support those research agendas. To ensure completeness and a measure of fairness, we compiled these results into a working paper and sent it back to all survey participants for feedback and corrections. Most participants agreed with our categorizations. A few added details. Table 2 gives an overview of our findings, including updates from survey participants.

Kiekintveld et al. [27] identify three key issues that a successful TAC SCM agent must address: dealing with substantial *uncertainty* in a highly *dynamic* economic environment, in competition with other self-interested agents whose behavior is naturally *strategic*. We observe a convergence between these issues and our independent findings from the TAC SCM questionnaire. Analysis of the questionnaire results shows how these issues, in conjunction with a variety of general research agendas, has driven the architectural styles adopted by the various teams.

Table 1
Teams participating in the TAC SCM architecture design survey.

Team	University	Team contact
Botticelli (B)	Brown University (USA)	Amy Greenwald Victor Naroditskiy
CMieux (CM)	Carnegie Mellon University (USA)	Michael Benisch Norman Sadeh
CrocodileAgent (CA)	University of Zagreb (Croatia)	Ana Petric Vedran Podobnik
DeepMaize (DM)	University of Michigan (USA)	Chris Kiekintveld Michael Wellman
Foreseer (F)	Cork Constraint Computation Centre (Ireland)	Kenneth Brown David Burke
Mertacor (M)	Aristotle University of Thessaloniki (Greece)	Pericles Mitkas Andreas Symeonidis
MinneTAC (MT)	University of Minnesota (USA)	John Collins Wolfgang Ketter
Southampton (S)	University of Southampton (UK)	Minghua He Nick Jennings
TacTex (TT)	University of Texas (USA)	David Pardoe Peter Stone
Tiancalli (T)	Benemerita Universidad Autonoma de Puebla (Mexico)	Darnes Ayala Daniel Galindo

3.1 Constraint optimization

A supply-chain agent situated in a trading environment has to comply with many internal and external constraints. These constraints apply to different parts of the supply-chain, such as procurement (e.g. reputation effect), production (e.g. limited production capacity), sales (e.g. can't sell more than effectively demanded by the market), and shipping (e.g. can't ship more than currently in the finished goods inventory). Nearly all the teams who answered our questionnaire applied constrained optimization in some way, so we have listed here the ones who highlighted it in their papers and the questionnaire response. The teams who focus on real time optimization, Botticelli [4], DeepMaize [26],

Table 2
Research agendas of teams and their architectural emphasis.

Research Agenda	Team	Architectural Emphasis
Constraint optimization	B, CM, F, MT	3rd party packages
	CM, DM	Internal optimization methods
Machine learning	CM, DM, MT, TT	External analysis framework, 3rd party packages
Dynamic supply-chain	CM, F, M, MT, T	Flexibility
Scalability	CA	Distributed Computation
Architecture	CA	IKB model for physical distribution
	MT	Blackboard architecture with evaluator chain
Empirical game theory	DM	External analysis framework
Decision coordination	CM, DM, M, S	Modularity
Dealing with uncertainty	B, F, S, MT	Modularity

Foreseer [8], MinneTAC [23] use mainly third party optimization packages, including CPlex [2], Ilog OPL [3], and lp_solve [4]. An exception is CMieux [3] which uses an internally-developed implementation of a search algorithm to solve a continuous knapsack problem for pricing customer offers.

3.2 Machine learning

Many agent designs depend on an external bootstrapping process to construct models and set parameters, using machine learning algorithms to learn from historical market data. Many agents have also some learning ability during operation to adapt to changing situations. Successful teams are generally those who perform a thorough off-line bootstrapping as well as online machine learning. CMieux [5], MinneTAC [23], and TacTex [30] identified the need to support learning and adaptation as primary concerns in the design of their agents. To support research agendas with a strong emphasis on machine learning, both CMieux [5] and TacTex [30] use the Weka [5] [39] machine learn-

[2] http://www.ilog.com/products/cplex/
[3] http://www.ilog.com/products/oplstudio/
[4] http://sourceforge.net/projects/lpsolve
[5] http://www.cs.waikato.ac.nz/ml/weka/

ing tool set. MinneTAC is using Matlab [6] in combination with the Netlab [7] neural network toolbox to develop and train market models, and to bootstrap the agent with the resulting models. At runtime, MinneTAC updates and adjusts those models using feedback and machine learning algorithms embedded in Evaluators (see Section 4.4) written in Java.

3.3 Management of dynamic supply chains

Traditionally, supply chains have been created and maintained through the interactions of human representatives of the various enterprises (component suppliers, manufacturers, wholesalers/distributors, retailers and customers) involved. However, the recent advent of autonomous trading agents opens new possibilities for automating and coordinating the decision making processes between the various parties involved. A good overview of multi-agent based supply chain management outside of TAC SCM is given in [10]. The TAC SCM simulation is an abstract model of a highly dynamic direct sales environment [12], as exemplified by Dell Inc. [8], for procurement, inventory management, production, and sales.

Given the dynamic nature of the TAC SCM environment and the challenges in the real world, many teams have a strong research focus on dynamic supply-chain behavior. These include CMieux [5], Foreseer [8], Mertacor [28,11], MinneTAC [24], and Tiancalli [15]. As a consequence of this research agenda the teams strive for high flexibility in their agent design, so that they can easily accommodate changes arising from new theory or from changes in the game environment itself. These responses are coherent with classical and contemporary literature in software engineering, which recognizes the central role of flexibility in software design for environments where ability to adapt to changing requirements is a dominant factor. This is indicative of an unsolved problem, since designing for high flexibility means that one doesn't know how to design it correctly.

3.4 Scalability

The CrocodileAgent [31] group at Zagreb is part of a larger group that is focused on autonomous agents for management of large-scale telecommunication networks. They view TAC SCM as an interesting challenge in building

[6] http://www.mathworks.com/
[7] http://www.ncrg.aston.ac.uk/netlab/
[8] `http://www.dell.com/`

an agent that can successfully operate in a dynamic, competitive environment, but they are also concerned with scalability and other issues that go far beyond the TAC SCM scenario. They have chosen to base their design on the JADE[9] [2] agent framework, since it has been well-proven for large-scale multi-agent situations with a huge number of entities on both the consumer, the business, or both sides of a supply chain (such as a telecom environment).

3.5 Architecture

CrocodileAgent [31] and Southampton SCM [18] have structured their agent decision processes around the the IKB model [36], a three layered agent-based framework for designing strategies in electronic trading markets. The first layer is the Information layer (I) which contains data gathered from the environment. The Knowledge layer (K) represents the knowledge extracted from the data stored in the information layer, and the Behavioral layer (B) encapsulates the reasoning and decision-making components that ultimately drive agent behavior. As reported by the CrocodileAgent [31] team, an advantage of using JADE is that the separation of I, K & B layers enables physical distribution of layers on multiple computers. In such a design, information layer agents parse out information from the TAC SCM server messages, while information and knowledge flows are implemented as JADE agent communication (ACL[10]-based messages). The separation of I, K & B layers and the introduction of JADE agent platform to the TAC SCM domain causes a much more complex system with lots of intercommunication, as reported by the CrocodileAgent team. CrocodileAgent deals with this overhead in TAC SCM, since their main agenda is to use JADE agents in their research regarding the next generation of telecommunication networks.

A MinneTAC agent is a component based framework. All data that must be shared among components are kept in the Repository, which acts as a blackboard [9]. For details on the MinneTAC design we refer the reader to Section 4. An interesting outcome of the questionnaire is that only the MinneTAC team has identified minimizing the learning curve for a researcher who wishes to work on a specific subproblem as an important design criterion.

3.6 Empirical game theory

The DeepMaize [27,37,19] group at Michigan pursues empirical game-theoretic analysis as one of their major research cornerstones. They employ an experi-

[9] http://jade.tilab.com/
[10] Agent Communication Language

mental methodology for explicit game-theoretic treatment of multi-agent systems simulation studies. For example, they have developed a bootstrap method to determine the best configuration of the agent behavior in the presence of adversary agents [19]. They also use game-theoretic analysis to assess the robustness of tournament rankings to strategic interactions. Many of their experiments require hundreds to thousands of simulations with a variety of competing agents. To support their work they have developed an extensive framework for setting up and running experiments, and for gathering and analyzing the resulting data [11] .

3.7 Decision coordination

Decision coordination is an important element of the research agendas for the DeepMaize [26], CMieux [5], MinneTAC [24], and Southampton [18] teams. This problem is commonly viewed as one of enabling independent decision processes to coordinate their actions in useful ways while minimizing the necessity to share representation and implementation details. This is important because of the difficulty in treating all the decisions an agent must make as a single problem. Indeed, real-world organizations often do a poor job of coordinating procurement and sales because they are functions of widely separated units within a typical industrial concern.

CMieux is the prime example of an agent that supports decision coordination explicitly in their design, incorporating a "strategy" module that dynamically sets product-mix and sales-volume targets in order to coordinate sales and procurement activities. MinneTAC uses a blackboard approach to allow decision processes to coordinate their actions through a common state representation, and Southampton uses the hierarchical IKB approach, in which the Knowledge layer of the IKB model could be viewed as a type of blackboard. DeepMaize [26] treats the combined decisions as a large optimization problem, decomposed into subproblems using a "value-based" approach. The result is that much of the coordination among decision processes is effectively managed by assigning values to finished goods, factory capacity, and individual components over an extended time horizon.

A particularly interesting approach to the decision coordination problem was taken by RedAgent [21], which used loosely-coupled "sub-agents" competing with each other in internal auction-based markets for finished goods, production capacity, and components. This achieved a radical decoupling of the various components, but proved to be uncompetitive after the game design was adjusted in 2005 to defeat some of the simplest approaches that lacked

[11] P. Jordan, private communication

adequate coordination among decisions. Specifically, agents that focused procurement only on keeping the factory in full production found themselves overproducing when the balance between factory capacity and expected customer demand was adjusted.

There are many different ways to coordinate the decision that every TAC SCM agent makes in support of sales, procurement, production scheduling, and sometimes inventory management. In this kind of setting it is advantageous to be able to replace individual decision-oriented components of an agent and compare their performances, e.g. compare two different sales modules on final profit. Many teams mentioned "modularity" as a separate goal for their designs, but we think that this is really a precondition that allows this sort of experimental flexibility.

3.8 Dealing with uncertainty

The TAC SCM competition scenario is designed to force agents to deal with uncertainty in many dimensions. Sodomka et al. [34] provide a good overview of the sources of uncertainty in the context of an approach to doing empirical study of agent performance. The Botticelli group clearly identifies the problem of dealing with uncertainty as one of their main research goals in [4]. SouthamptonSCM [18] employs a bidding strategy that uses fuzzy logic to adapt prices according to the uncertain market situation. SouthamptonSCM told us in the questionnaire that their software package for fuzzy reasoning on price adaptation will be released soon.

The key architectural decision in Foreseer [8] is that all constraint optimization models used in the agent are subject to uncertainty. In Foreseer both the customer bidding model and the component procurement model are subject to uncertainty. Both are parameterized, such that probability distributions representing the current *belief* of the state of the market can be passed into and used by the models. Given the uncertainty of the market, these beliefs allow Foreseer to represent different possible market states with different probabilities.

MinneTAC [24] observes market conditions to characterize the microeconomic situation of the market, economic regimes, and to predict future market situations. MinneTAC maintains a distribution of predicted economic regimes and uses this information to make both tactical decisions, such as pricing, and strategic decisions, such as product mix and production planning.

3.9 Published TAC SCM designs

Several participants in TAC SCM have described their agent designs. He *et al.* [18] have adopted a design consisting of three internal "agents" to handle Sales, Procurement, and Production/Shipping. Sales decisions use a fuzzy logic module. Some algorithmic details are given, but there is little further detail on the architecture of the agent. TacTex05, the winner of the 2005 competition [30] is based on two major modules, a Supply Manager that handles procurement and inventory, and a Demand Manager that handles sales, production, and shipping. These modules are supported by a supplier model, a customer demand model, and a pricing model that estimates sales order probability.

The overall survey outcome shows that there are common themes emerging from the different research groups on how to design a successful agent architecture. These include common software engineering quality criteria, such as modularity, low coupling, and separation of concerns, in addition to more problem-specific approaches, such as coordination of sales and procurement through internal models of inventory and prices, and assigning current and future value to inventory and production resources. There are also some strong differences, such as how to organize the communication between the different modules and which modules should own the data for specific tasks. These findings, and the fact that after several years of competition there is still much to be learned, suggest that the recipe for a full competent supply-chain trading agent is still an unsolved problem, even for an abstract, constrained environment like TAC SCM. In summary we can say that agents designed for highly dynamic, uncertain, or complex environments must be capable of flexible autonomous actions [40].

4 The design of MinneTAC

The MinneTAC agent is intended to be both a research vehicle and a teaching vehicle. As a research vehicle, it must be able to compete effectively in the annual TAC SCM tournaments, and it must provide an array of features to support a research agenda, including easy reconfiguration and good data collection capabilities. As a teaching vehicle, it must be an example of good design, and it must make it easy for students to make substantial contributions within the scope of a one or two semester project.

The design of MinneTAC follows a component-oriented approach [35]. The idea is to provide an infrastructure that manages data and interactions with the game server, that cleanly separates behavioral components from each

other, and that allows easy substitution of different implementations for the major decision processes. This allows individual researchers to encapsulate agent decision problems within the bounds of individual components that have minimal dependencies among themselves. The result is a system that is easy to modify and configure, but there is an essential tension between a highly modular structure and the need for effective integration of the various decision processes. For example, sales and procurement decisions must be coordinated so that the parts that are purchased can be assembled and sold at a profit, and so that inventories can be replenished as they are depleted by sales. We address this tension with two design features: a common data store that contains the agent's global state, and a set of configurable evaluation modules that are shared among the decision processes. A specific example is given in Section 5.3.

Two pieces of software form the foundation of MinneTAC: the Apache Excalibur component framework [14], and the "agentware" package distributed by the TAC SCM organizers. Excalibur provides the standards and tools to build components and configure working agents from collections of individual components, and the agentware package handles interaction with the game server.

4.1 A brief overview of Excalibur

Apache Excalibur is a general-purpose framework for building highly configurable systems out of independent components. It is widely used as a foundation for middleware and for server software, such as the OpenORB CORBA implementation [12] and the Cocoon web application framework [13], 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.

Excalibur components are independent entities, in the sense that they typically have very few dependencies on each other, and minimal, well-defined dependencies on the Excalibur framework itself. Components are coarse-grained entities, each typically composed of a number of classes. Control inversion puts primary control in the Excalibur "container", which loads components, sets

[12] OpenORB.sourceforge.net
[13] cocoon.apache.org

up logfiles, configures the components, and starts any components that run independent threads.

Each Excalibur component is designed to fulfill a specific *role*, and an Excalibur system is a set of roles, each of which is mapped to a specific Java class. A role has a name, a set of responsibilities, and an interface specification. An Excalibur application is composed of the Excalibur infrastructure, a container that initializes the system, and the components specified by the configuration. Configuration files specify the roles, the classes that satisfy those roles, and configuration parameters for those classes. The framework reads the configuration files, loads the specified classes, and invokes the Excalibur interfaces on each component.

4.2 MinneTAC architecture

Following Bass et al. [1], we use the term "architecture" to refer to the set of components that make up our system, along with their properties and relationships. A MinneTAC agent is a set of components layered on the Excalibur container, as shown in Figure 1. In the standard arrangement, four of these components are responsible for the major decision processes: Sales, Procurement, Production, and Shipping. In some configurations, an LpSolver component is included to provide optimization services, and in some, the Procurement module is split in two, separating the problem of formulating RFQs for suppliers from the problem of deciding which supplier offers to accept. All data that must be shared among components is kept in the Repository, which acts as a blackboard [9] and is visible to all other components. The Oracle hosts a large number of smaller components that maintain market and inventory models, and do analysis and prediction. The Communications component handles all interaction with the game server. By minimizing couplings between the components, this architecture completely separates the major decision processes, thus allowing researchers to work independently. Ideally, each component depends only on Excalibur and the Repository.

The agent opens four configuration files when it starts. Two of them are interesting in the context of this paper. The system configuration file specifies the set of roles that make up the system, along with the classes that implement those roles. This allows the major components (Sales, Procurement, Production, Shipping) to be swapped out with a simple edit. The component configuration file specifies runtime configuration options for each component. For example, the Sales component may have a parameter that controls the maximum level of overcommitment of its existing inventory or capacity when it makes customer offers. More importantly, the various Evaluators are specified and configured in this file (see Section 4.4.1).

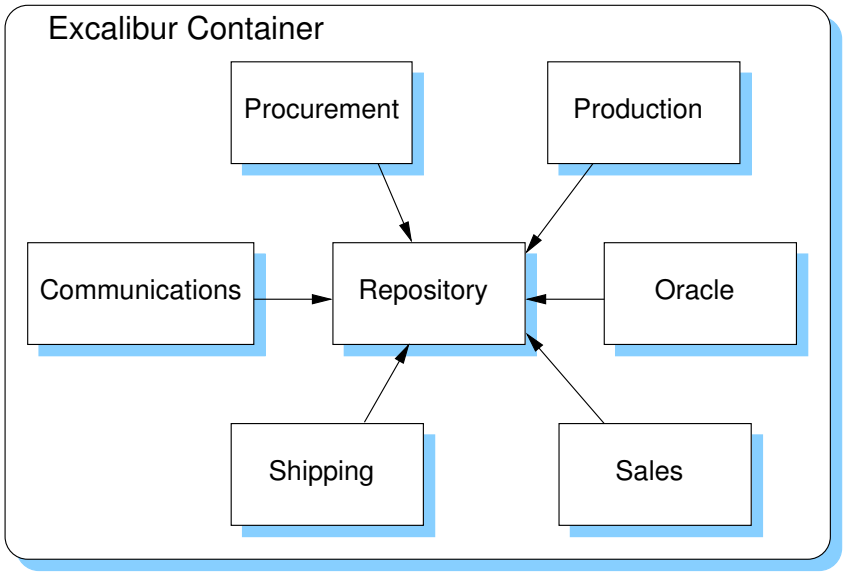

Fig. 1. MinneTAC Architecture. Arrows indicate dependencies.

In the following sections, we provide more detail on the three "core" components of MinneTAC, the COMMUNICATIONS, REPOSITORY, and ORACLE components. The other components implement the various decision processes, and should be thought of as "role interfaces" since multiple implementations exist for each of them. To flesh out this concept, we also provide in Section 5 an in-depth look at two of our Sales component implementations, the price-driven sales manager used in the 2005 and 2006 competitions, and the quota-driven sales manager used in the 2007 competition, as well as a brief overview of one of our procurement implementations.

4.3 Communications

A TAC SCM Agent is a "reactive system" in the sense that it responds to events coming from the game server [38]. 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.

Figure 2 shows the communication activity for a typical 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

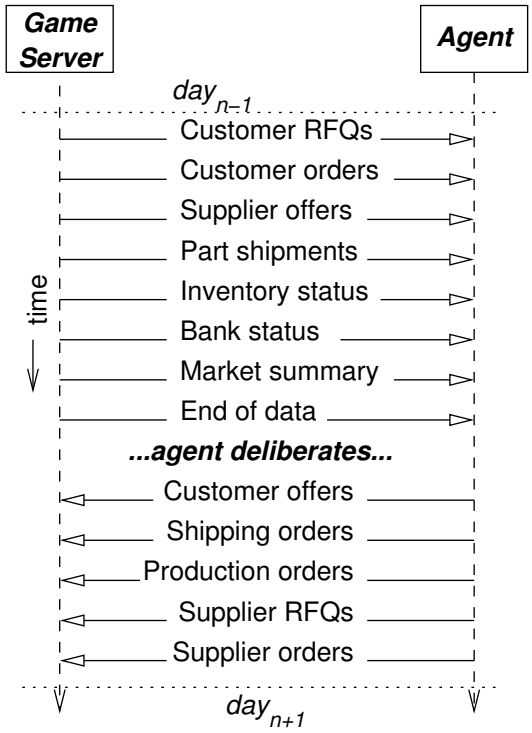

Fig. 2. One day of communications activity between the game server and an agent.

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, and it may issue additional supplier RFQs each day. The length of a simulation day is fixed by the server; in the standard tournament configuration, days are 15 seconds long. If the agent does not return its messages within this interval, they are lost.

4.4 Repository

The REPOSITORY is the one component that is visible to all the other components. From a software architecture standpoint, the REPOSITORY plays the part of the Blackboard in the *Blackboard* pattern [9]. The general idea of a Blackboard system is that data and partial solutions are shared among a number of "knowledge sources" that can access and update the central blackboard when they have something to contribute. The ORACLE and its Evaluators (see Section 4.4.1) are the primary knowledge sources, adding Evaluations to the data elements in the REPOSITORY. The decision components (SALES, PROCUREMENT, PRODUCTION, and SHIPPING), responding to REPOSITORY state-transition events, drive the process indirectly, by requesting Evaluations, and by recording their decisions on the blackboard. We now explain in more detail how this works, using two major features of the REPOSITORY: *events* and *evaluations*.

Events. As shown in Figure 2, 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 a special message, the end-of-data message 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, retrieving data and evaluations from the repository, and recording the results of their decisions back into the repository. Finally, the resulting decisions, in the form of RFQs, offers, orders, and daily production and shipping schedules, are retrieved from the REPOSITORY by the COMMUNICATIONS component and sent to the server.

Figure 3 shows state transitions and their associated events. When a component receives the *data-available* event, it is able to inspect the incoming data for the day's transactions and perform whatever analysis is needed to update its models. When a component receives the *decision* event, it is expected to finalize its decisions and record its outgoing messages back in the REPOSITORY.

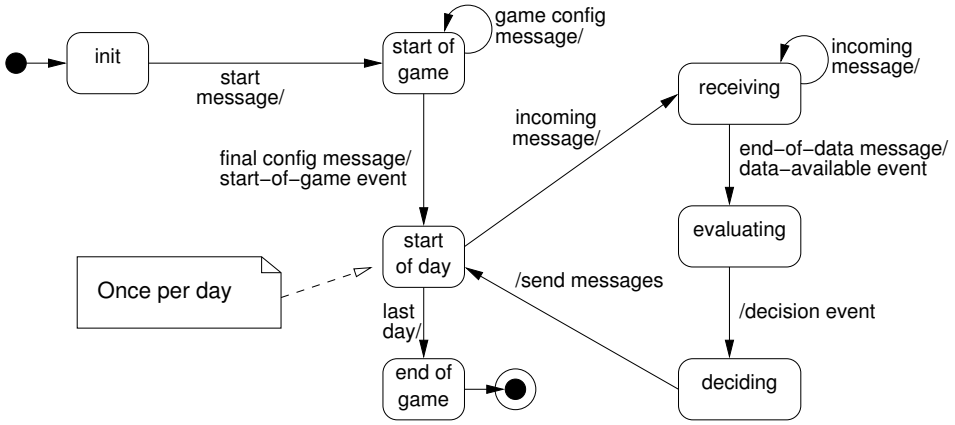

Fig. 3. States and transitions in the REPOSITORY component. Arcs are labeled with event/action pairs.

In this interaction, the REPOSITORY acts as a Subject and the other components as Observers in the *Observer* pattern [16]. An important limitation of the Observer pattern is that the sequence of notifications is not controlled, although in most implementations it is repeatable. But the order of event processing is important for the MinneTAC decision processes. For example, it greatly simplifies the Sales decision process to know that the current day's Shipping decisions have already been made. To allow event sequencing without introducing new dependencies, two events are generated by the REPOSITORY for each day of a game. The *data-available* event is a signal to read the incoming messages and do basic data analysis. The subsequent *decision* event

is a signal to make the daily decisions and post the outgoing messages back in the REPOSITORY. The decision event itself provides an additional level of sequence control by allowing components to "refuse" the event until one or more other components (identified by role names) have made their decisions. Components that have refused the event will receive it again once all other components have had an opportunity to process it. To ensure that SALES decisions are made after SHIPPING decisions, SALES must refuse to accept the decision event until after it sees "shipping" among the roles that have already processed it. No additional dependencies are introduced by this mechanism, since the role names are simply added to the event object itself, and the names come from the component configuration file, not the code.

Evaluators. As indicated earlier, a goal of the MinneTAC design is to minimize coupling between the various components, while providing appropriate tools to coordinate decisions. How can the decision components communicate, if they cannot depend on one another? Our approach is to use *evaluations* that are accessible through the various data elements in the REPOSITORY. The general idea is that when a component needs to make a decision (for example, SALES composing its bids for customer requests or PROCUREMENT generating its supplier requests), it will inspect the available data, update estimates of trends and probabilities, and possibly run some utility-maximizing function. The available data consists of any data maintained internally by the component, 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 thereby are made available to all components. In MinneTAC, these analyses are implemented by the ORACLE component through a configurable set of *evaluators*.

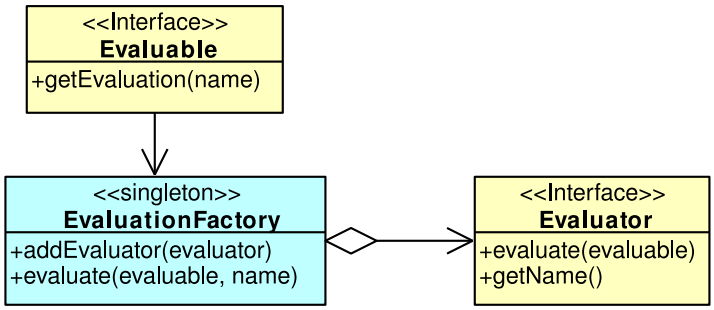

Fig. 4. Evaluables and Evaluators.

All the major data elements in the REPOSITORY (such as RFQs, offers, orders, products, components, market reports, etc.) are Evaluable types. As shown in the UML class diagram in Figure 4, each Evaluable can be queried for associated Evaluations, by passing it the name of the desired evaluation. The task of mapping names to Evaluator instances is delegated to an EvaluationFactory, which maintains a mapping of Evaluation names to Evaluator instances, and is responsible for producing Evaluations when they are requested. It does

this by inspecting the class of the Evaluable and the name of the requested Evaluation, and invoking the appropriate *evaluate* method on an associated Evaluator, as shown in Figure 5.

Evaluations are *composable*. Evaluators implement back-chaining by requesting other Evaluations in the process of producing their results, and therefore many Evaluations are composed from other, presumably simpler, Evaluations. Evaluators are hosted by the ORACLE component, which is responsible for loading and configuring Evaluators. Evaluators are registered with the REPOSITORY when they are configured, thus making them known to the EvaluationFactory. Because a given Evaluation may be requested multiple times during the agent's decision process, most Evaluators cache their results as long as they have reason to believe they remain valid. In most cases, Evaluator results remain valid until the next data-available event arrives.

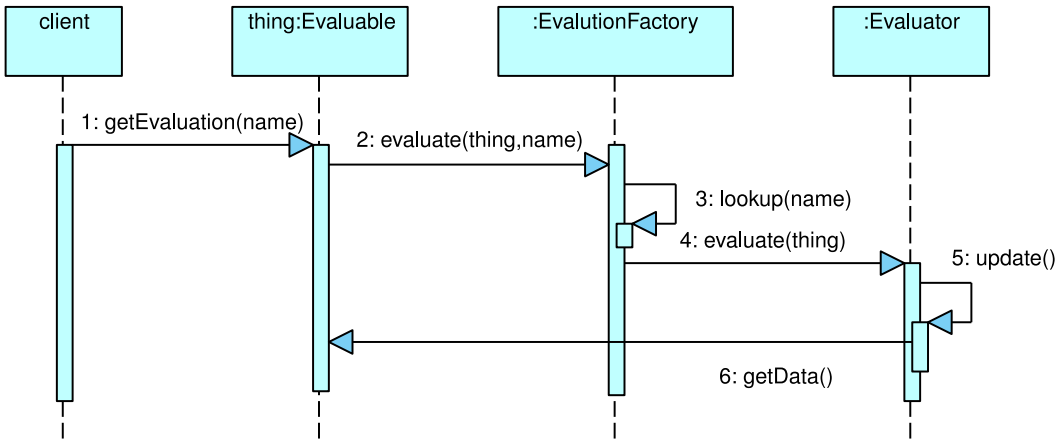

Fig. 5. Processing an evaluation.

Figure 6 shows a simple example of some Evaluable instances and a set of Evaluations that might be associated with them. The *price* evaluation might combine parts cost information with an estimate of current market conditions. The *profit* evaluation could compute the difference between parts cost and *price*. The *sort-by-profit* evaluation would need the *profit* evaluations on the individual RFQs. Extended examples of our application of this mechanism will be given later in Section 5.

4.4.1 Oracle

The ORACLE component is essentially a meta-component, since its only purpose is to provide a framework for a set of small configurable components that may be used to perform analysis and prediction tasks. Most of these are Evaluators, but a few other types are supported as well. The ORACLE itself simply reads its configuration data, and uses it to create and configure instances of Evaluators and other subclasses of ConfiguredObject. The top-level

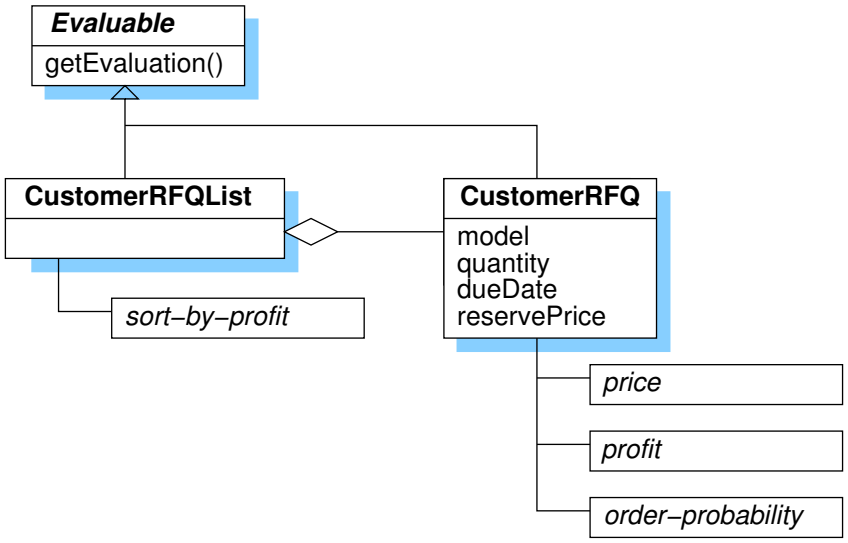

Fig. 6. RFQ evaluation example. CustomerRFQList and CustomerRFQ are Evaluable instances, and the small rectangles represent Evaluations that might be associated with them.

classes within the ORACLE component are shown in the UML class diagram in Figure 7.

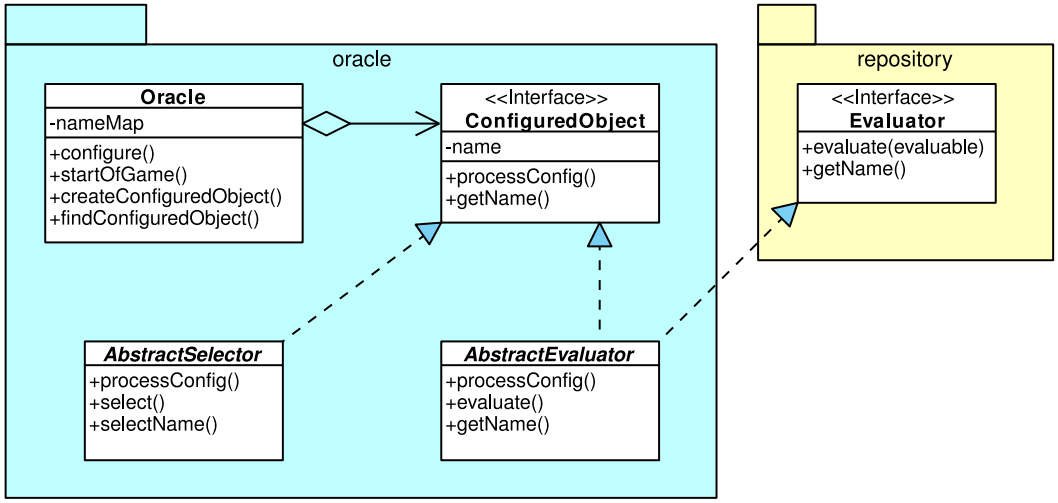

Fig. 7. Principal classes in the ORACLE component.

ConfiguredObject is an abstract class whose instances have names and an ability to configure themselves, given an appropriate XML clause. The ORACLE creates ConfiguredObject instances and keeps track of them by mapping their names to instances[14]. When it starts, the ORACLE processes a configuration clause that typically includes at least two subclauses. The first is a `<setup>` clause, which is processed at the time the ORACLE is created, during agent

[14] This is a separate mapping from the one maintained by the EvaluationFactory in the REPOSITORY (see Section 4.4). It serves a different purpose, and not every ConfiguredObject is accessible outside the ORACLE, as Evaluators are.

initialization. At this time, objects can be created that do not need access to game parameters. A typical example is a model element that must read its initialization data from a file or database that has been created off-line, perhaps by analyzing prior games using machine learning techniques [22]. This must be done before the game begins simply because of the time required to set up these models; once the game begins, the agent must complete its work in less than 15 seconds each day. Other clauses are processed by the ORACLE after the start-of-game event has been received, thus allowing objects to access game parameters from the REPOSITORY when they are created. For example, many evaluators need to initialize themselves using data from the server's Component Catalog or Bill of Materials, which are sent to the agent at the start of a game.

Figure 8 illustrates the configuration clause for an Evaluator called "order-probability" that estimates the sales order probability for each product (the *order-probability* evaluator in Figure 9) by combining a median price estimate with a slope estimate. By convention, the output of any evaluator that promises to estimate order probability is an object called a `Pricer` that has two methods: `getPrice()` returns the predicted median price, and `getPriceForProbability(p)` returns the price corresponding to the given probability p. Inputs to this evaluator are two other evaluators, named "median-price" and "slope-estimate." The implementations of these evaluators are in turn specified in other configuration clauses. Indeed, several different methods of estimating median price and slope can be tested and compared simply by changing the class associated with the name in the configuration file.

```
<evaluator class="edu.umn.cs.tac.oracle.eval.LinearOPEstimator"
           name="order-probability">
    <inputs>
           <median source="median-price" />
           <slope source="slope-estimate" />
    </inputs>
</evaluator>
```

Fig. 8. Configuration clause for an order-probability estimator that uses a median price and a slope estimate as data sources

The most common subclasses of ConfiguredObject are Evaluators and Selectors. We have discussed Evaluators at length in Section 4.4, and we shall see an extended example of their use in the next section. A Selector is simply a switch that can be used to select different models or evaluators in different game situations. For example, the early part of a game is typically characterized by customer prices that start high and fall rapidly as agents acquire parts and begin building up inventories. Later in the game, prices are less predictable and more sophisticated models may be useful. A simple DateSelector can be used to switch between pricing models at a particular preset date, or

a more sophisticated Selector could be used to switch models once the initial price decline bottoms out. The selector is configured with the two models and a mapping of dates to models. A call to its `select()` method returns the correct model for the current date. An interesting subtype of Selector is Mixer, which blends one model into another over a period of time, thereby eliminating sharp transitions. This can be important when feedback loops are being used to fine-tune model prices to the current game environment, as described in the following section.

5 Examples

Here we describe in some detail three examples of the use and configuration of evaluators in support of significant agent decision processes. The first is the SALES component that was used in the 2006 Trading Agent Competition, in which MinneTAC placed sixth overall. The second is the 2007 revision of the SALES component, and the third is the PROCUREMENT component that ran in the 2007 Trading Agent Competition.

5.1 Price-driven Sales

To illustrate the power of the MinneTAC design and its use of Evaluators, we show in Figure 9 the evaluation chain that is used to produce sales quotas and set prices in a relatively simple MinneTAC configuration. Each of the cells in this diagram is an Evaluator. A version of the SALES component called PriceDrivenSalesManager is conceptually very simple – it places bids on each customer RFQ for which the randomized-price evaluator returns a non-zero value. The core of this chain is the allocation evaluator, which composes and solves a linear program each game day that represents a combined product-mix and resource-allocation problem that maximizes expected profit. The objective function is

$$\Phi = \sum_{d=0}^{h} \sum_{g \in \mathcal{G}} \Phi_{d,g} A_{d,g} \tag{1}$$

where Φ is the total profit over some time horizon h, $\mathcal{G}$ is the set of goods or products that can be produced by the agent, $\Phi_{d,g}$ is the (projected) profit for good g on day d, and $A_{d,g}$ is the allocation or "sales quota" for good g on day d. The constraints are given by evaluators *available-factory-capacity*, the current day's *effective-demand*, projected *future-demand*, and by REPOSITORY data, such as existing and projected inventories of parts and finished products, and

outstanding customer and supplier orders. Predicted profit per unit $\Phi_{d,g}$ for each product type is the difference between Evaluations called *median-price* and *cost-basis* for those products.

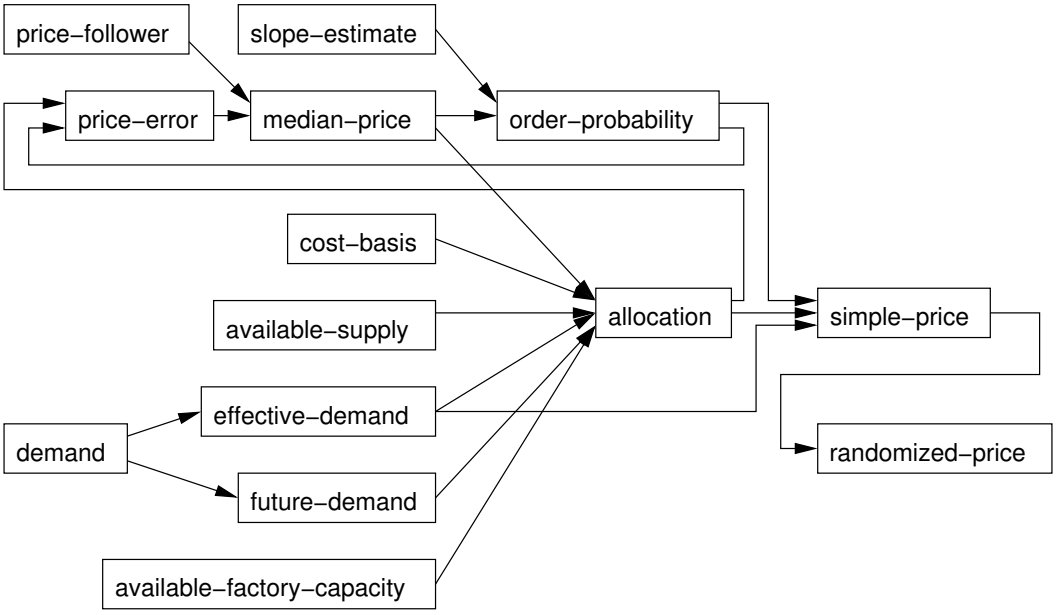

Fig. 9. Evaluator chain for sales quota and pricing.

The Evaluation generated by the *allocation* evaluator gives recommended sales quotas for each product over a time horizon. Given a sales quota for a given product, the demand for that product, and an *order-probability* function, the *simple-price* evaluator computes a price that is expected to sell the desired quota, assuming that price is offered on all the demand for that product. In other words, if the quota is 25 units and the demand is for 100 units, *simple-price* computes a price that is expected to be accepted by only 25% of the customers. Since there is some uncertainty in the predictions of price and order probability, randomized-price adds a slight variability to offer prices. This improves the information content and reduces variability of the returned orders.

Market prices are tracked by a *price-follower* evaluator, which observes the daily high prices reported by the server. The *price-follower* implements a straightforward double-exponential smoothing function

$$price_d^{sm} = \alpha(price_d^{max}) + (1-\alpha)(price_{d-1}^{sm} + trend_{d-1}) \quad (2)$$

$$trend_d = \gamma(price_d^{sm} - price_{d-1}^{sm}) + (1-\gamma)trend_{d-1} \quad (3)$$

where $price_d^{max}$ is the observed high price for a given product on day d (the highest price at which the product was sold on the previous day), $price_d^{sm}$ is the smoothed price for the current day d, $trend_d$ is the trend for the current day, and α and γ are the smoothing parameters. The resulting smoothed price

estimate is too high to support sales, since it is tracking the daily high price, and it depends strongly on the detailed behavior of other agents. Therefore, we use a feedback mechanism to adjust our price estimates, as shown graphically in Figure 10. Each day d, the *order-probability* evaluator generates a pricing function $P_d(order|price)$. $price^{est}$ is the price computed for yesterday's sales quotas Q_{d-1}. Yesterday's observed price $price^{obs}$ is computed by applying yesterday's pricing function to today's customer orders O_d. Recall that orders from customers arrive the day after the offers were made. The correction computed by *price-error* is the difference between estimated and observed prices

$$err_p = price^{obs} - price^{est} \tag{4}$$

The *median-price* evaluator then computes a median price

$$price^{med} = price^{sm} + err_p \tag{5}$$

for the current day, giving the corrected output of *price-follower*.

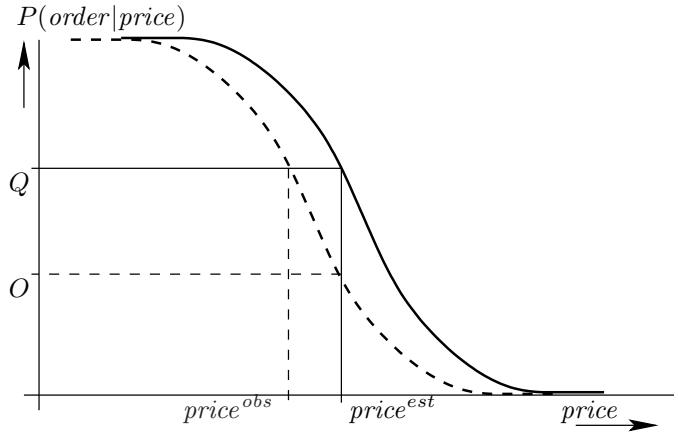

Fig. 10. Estimating actual market price $price^{act}$, given sales quota Q, order volume O and an estimate of the order probability function P.

We have reported elsewhere [22] a different MinneTAC sales price model based on Gaussian Mixture Models built out of evaluators. In that model the *price-follower*, *median-price*, and *order-probability* evaluators shown in Figure 9 have been replaced, and the *slope-estimate* evaluator omitted. Two ConfiguredObject types were added to load training data for the model when the agent starts. This is the configuration that ran in the 2006 TAC SCM tournament; the details are beyond the scope of this paper.

In reality, the outline of the price-driven SALES component given in Section 5.1 is significantly oversimplified. This is because the uncertainty in price and order-probability estimates can cause over-selling or under-selling against a quota. Over-selling can be a significant problem when the actual sales volume violates inventory or capacity constraints, with the result that the agent is unable to ship customer orders on time. Late shipments are subject to substantial penalties, and therefore the price-driven sales component avoids overcommitment through detailed accounting of commitments against inventory constraints. The result is that the agent frequently fails to make enough offers to meet its intended sales quotas, in return for avoiding late-delivery penalties. This could be corrected by increasing prices slightly to account for a higher order/offer ratio, but this design computes prices for the original quotas. The price-adjusting feedback mechanism works on the assumption that the offered price corresponds to the original quota/demand ratio, and this assumption is frequently violated. The resulting variable difference between intended and actual order/offer ratios introduces noise and reduces the effectiveness of the feedback.

The MinneTAC configuration for 2007 took a somewhat different approach. Instead of avoiding any possibility of overcommitment, it avoids overcommitment in expectation, with an adjustable risk tolerance. The idea is that if we know something about the probability of overcommitment, we can keep that probability under control. Figure 11 shows a new evaluator feedback loop that was added to the 2007 configuration.

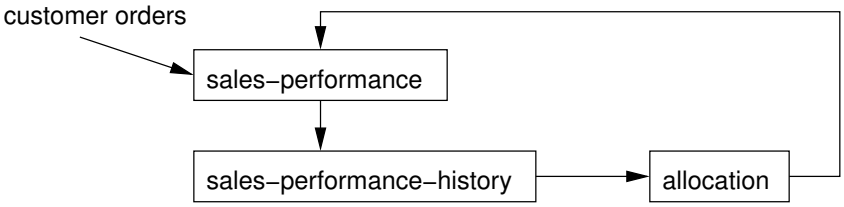

Fig. 11. Evaluator feedback chain for managing overcommitment.

In this configuration, the *sales-performance* evaluator compares quotas for the previous day with orders on the current day, producing a ratio for each product. This information is smoothed using a decaying rolling average in *sales-performance-history*, which generates mean and standard deviation data for the recent past. Under this scheme, *allocation* adjusts each of its current-day sales quotas $A_{d,g}$ as

$$A'_{d,g} = \min(A_{d,g}, \frac{constraint_{inv}}{\overline{perf}_g + rt \cdot \sigma_g}) \tag{6}$$

where $A_{d,g}$ is the quota value for product g produced by the linear program described in Section 5.1, $constraint_{inv}$ is the inventory constraint that (potentially) limits the value of $A_{d,g}$, $\overline{perf_g}$ is the rolling mean of sales performance for g, σ_g is the standard deviation of the recent performance history, and rt is a risk-tolerance parameter (typically set to 0.5). The actual computation is a bit more complex than this, because inventory constraints due to individual parts can apply to multiple products that require those parts, and so the constraint is typically on the sum of the product allocations associated with any particular part.

We compared the performance of the 2006 and 2007 configurations with regard to this modification by computing the ratios of offers to quotas, orders to quotas, and orders to offers. As we can see in Table 3, the distributions of these ratios are strongly skewed. This is evidenced by the large difference between the mean and median values, and by the large standard deviations. There are a few very large outliers in both data sets. These can happen in situations with small quotas and high demand, if MinneTAC's price estimate is too low. Therefore we compared them with the Wilcoxon-Mann-Whitney two-sample rank-sum test. The p column in Table 3 represent the (two-tailed) probability that the corresponding data from the 2007 and 2006 games are from the same distribution. It seems likely that the change from certain avoidance of overcommitment to avoidance in expectation has changed the behavior. In fact, the penalties paid due to overcommitments for the 2007 agent were under 1% of revenue, while the penalties for the 2006 agent were only slightly lower, about 0.8%. That is because the 2006 agent used absolute control of overcommitment only with respect to inventory, and we experienced occasional overcommitment of production capacity.

Table 3
Performance comparison of the 2006 and 2007 MinneTAC configurations.

	MinneTAC 2006			MinneTAC 2007			
	mean	median	σ	mean	median	σ	p
offer/quota	3.41	1.33	8.86	4.43	1.35	12.47	< 0.01
order/quota	2.05	0.91	7.17	1.84	0.98	6.77	< 0.01
order/offer	0.58	0.80	0.44	0.57	0.69	0.40	< 0.01

This data comes from a set of games in the final rounds in the 2006 competition, and from the quarter-final rounds in the 2007 competition. We have omitted data from the first 10 days and the last 10 days of each game, because those data tend to be dominated by the extreme instability of price estimates at the beginning and end of each game. The comparison is complicated by several factors, including the high variability in the game environment, and the fact that the 2006 data is from a different set of games, with a different set of competitors, compared to the 2007 data. However, we are considering

only an internal performance measure that should be relatively unaffected by the competition or by the statistics of particular games. Because the 2007 agent is presumably doing a better job of matching its optimized quotas to its actual sales behavior, we would expect the 2007 agent to exhibit more aggressive selling, as evidenced by higher ratios of offers to quotas and of offers to orders. However, assuming the price-tracking feedback is working well, we should see smaller differences in the relationship between quotas and orders. This is in fact what we observe. A more complete comparison of these configurations, one that will allow us to show the impact of this small change on agent profitability, will be performed using the method described by Sodomka et al. in [34]. This method works by controlling the random number generators in the game server that control customer demand, supplier capacity and other sources of variability, thereby allowing the same game to be run with different agents or agent configurations.

5.3 Procurement

Procurement decisions in a supply-chain trading agent must balance a number of factors aside from ensuring that components are available to the manufacturing operation when they will be needed, and that sales is provided with products it can sell at a profit. In supply-chain situations generally, it is common for prices to be lower for longer leadtimes and larger commitments, but procurement must balance procurement cost against the cost to hold inventory, and against the cost of acquiring and holding inventory that is not selling well in the customer market. In many environments, including TAC SCM, reputation is also an issue. If suppliers are repeatedly asked to bid on large quantity orders, and then the offers are rejected, suppliers will likely raise their offer prices as a way of discounting the value of the uncertain business.

In Figure 12, we show the evaluator chain that drives procurement decisions in MinneTAC. Note that the evaluator in the upper-left corner is *allocation*, which we saw previously in Figure 9. This and the *future-demand* estimate are the core elements of coordination between Sales and Procurement, informing the Procurement decision process of what Sales would like to sell, while informing Sales of the most profitable way to use the components that Procurement has made available. Note that this approach risks a bad positive feedback loop. If inventories fall short, then sales quotas will fall, and procurement will receive a signal to purchase fewer parts. This problem is avoided by using the *safety-stock-monitor* to override the resulting low quotas when they are caused by inventory shortages. An alternative method that is being explored for the 2008 design is to run the *allocation* optimization twice, once with inventory constraints to drive sales, and once without inventory constraints to drive procurement.

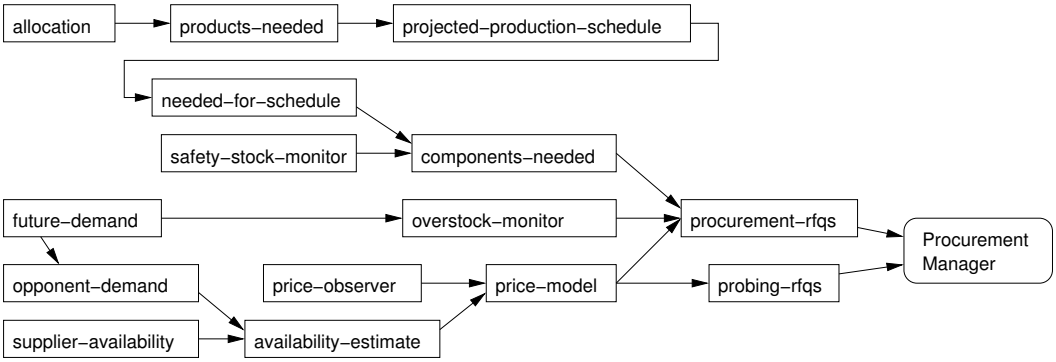

Fig. 12. Evaluator chain for procurement in MinneTAC.

The key elements of the procurement decision process are the *price-model* and estimates of needed supply over both short-term and longer-term time horizons. The *price-observer* evaluator monitors all interactions with suppliers that might produce price signals, and supplies a preliminary price model based on a weighted nearest-neighbors method similar to that used by Benisch et al. in the 2005 CMieux agent [5]. Some of those signals are produced from price-probes – simple requests to suppliers for future price estimates that carry no commitment to purchase. Probes are generated by *probing-rfqs* when the price model's data quality falls below a threshold, generally due to poor coverage of a potentially interesting future time period. Since supplier pricing in TAC SCM is a well-known function of demand and supply, another element of the pricing model is the *availability-estimate*, which estimates supplier capacity from information about expected future *opponent-demand* and from *supplier-availability* data derived from observed prices, and from situations where a supplier is unable to fulfill a request due to lack of capacity.

Demand for components must be estimated over a relatively long horizon in order to achieve good prices in the procurement market. Part of that information comes from the *allocation* module that we discussed in Section 5.1, and part comes from longer-term estimates based on general knowledge of the game environment. These are combined in *products-needed*. The capacity limits of the production facility are factored in by *projected-production-schedule*, and the conversion from products to parts is made by *needed-for-schedule*. This is combined with inventory monitoring from *safety-stock-monitor* to produce an integrated view of future component requirements in *components-needed*. Finally, *procurement-rfqs* combines future component needs with the price model and a second inventory monitor *overstock-monitor* that is focused on minimizing end-of-game inventory to produce sets of RFQs that are expected to procure the needed components at the lowest possible price.

6 Evaluation

The software architecture of MinneTAC is strongly focused on strict control of dependencies, and on flexible configuration. We evaluate the success of this design by asking three questions:

(1) Does the agent perform well, and to what extent does the design affect agent performance?
(2) How does the design rate on objective measures of design quality?
(3) Does the design meet the "usability" challenges described in Section 2.2?

6.1 Performance

There are two measures of agent performance that could be affected by the design. We will start by evaluating overhead: Does the design impose an unacceptable runtime overhead? We can also measure how well the agent performs in competition against other agents that have been implemented with different designs.

The Excalibur 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 (with a simple Sales component) 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. A Sales component that relies on solving a linear program each round takes longer, but its performance is almost entirely determined by the time required to compose and solve the linear program.

MinneTAC has done reasonably well in the official TAC SCM tournaments since 2003. In 2005 and 2006 it was a finalist. Each year, MinneTAC has been fielded with a new implementation of at least one of the decision components (Sales, Procurement, Production, and Shipping), and several others have been implemented but have not been entered into the competition. The ease with which these new components could be implemented and configured into the agent is a testament to the design we describe here.

Mitch Kapor says that software design bridges the world of people and human purpose with the world of technology [20]. 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 users of a software design are the programmers who implement it and extend it. The programmer who must change it later is perhaps even more strongly impacted. From the implementer's standpoint, good design is easy to understand, easy to implement correctly, and easy to maintain. From the maintainer's standpoint, a good design is one that is easily understood, and that can sustain change without losing integrity.

A great many qualitative and quantitative metrics have been proposed to measure the quality of software design [7,17,6]. Unfortunately, most of them do not produce absolute results but rather comparisons between alternative designs, and most of them are not directly applicable to component-oriented systems such as MinneTAC in which the primary design artifacts that we wish to measure are multi-class components rather than individual classes. One quality metric that is applicable is Martin's coupling metrics [29] using afferent coupling C_a to measure dependencies of other components on the elements of a given component, and efferent coupling C_e to measure dependencies of a component on other components. In an ideal system, each component has either $C_a = 0$ or $C_e = 0$, which means that each component is either a source or a sink of dependencies, but not both. In such a design, it is relatively easy for developers to determine the likely results of changes, and components for which $C_a = 0$ can be freely modified without concern for violating assumptions that might be made by developers of other components.

Table 4
Design quality metrics for MinneTAC components.

Component	C_a	C_e	I
Sales	0	1	1
Procurement	0	1	1
Delivery	0	1	1
Production	0	1	1
Communication	0	1	1
Oracle	0	2	1
LpSolver	1	0	0
Repository	6	0	0

In Table 4 we see the Martin dependency metrics for the 2007 MinneTAC

configuration, as determined by the jDepend tool[15]. This configuration includes an additional component to encapsulate the lp_solve linear programming solver, used by an evaluator that computes sales quotas as described in Section 5.1. The final column gives the "Instability" measure $I = C_e/C_e + C_a$, which is intended to indicate the systems resilience to change originating in that component. A score close to zero indicates that the stability of the entire system depends on the stability of that component, while a score close to one indicates that the system's stability does not depend on stability of that component. In the case of MinneTAC, the LpSolver component is simply a wrapper for a stable third-party package, and the REPOSITORY contains essentially the entire state of the agent. Both have been extremely stable over the last five years, while many different versions of the other components have been introduced without worry about affecting the viability of the agent.

Of course, there is another sort of dependency in the design of MinneTAC that is not captured by standard design-quality metrics. That is the dependency of the behavior of decision components on the detailed behavior of specific evaluators, and the need for effective coordination of the various decision processes. The good news is that researchers can freely experiment with these behaviors and coordination schemes without concern for whether the agent will build and run correctly.

6.3 Usability

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.

There is considerable evidence that our design has met its goals.

- Inexperienced student programmers have been able to contribute significant functionality without needing to understand the entire system. Examples include two different SHIPPING components, two different PRODUCTION components, five different SALES components, six different PROCUREMENT components, and over 80 different Evaluators, written by 22 students over a period of four years. Most of them worked on MinneTAC for just one semester.

  One undergraduate designed, built, and tested a complete procurement manager with active supplier price modeling, consisting of 11 evaluators and a procurement component, with a total of about 4600 lines of code, in a semester. This was possible because the simple structure of evaluators

[15] `http://clarkware.com/software/Depend.html`

makes it easy to decompose complex decision processes into small pieces with simple structure.

- The standardized log-message format produced by the Excalibur infrastructure makes data extraction relatively easy, even though MinneTAC generates approximately 5Mb of data for a typical game. A wide variety of analyses have been carried out with this data. An example of such an analysis is given in [25], where we were able to show that the original design of the game gave a large advantage to the agent that won a procurement lottery on the first day of the game.
- Selectors and Mixers (see Section 4.4.1) are very recent additions to the MinneTAC design, and they add considerable expressive power for constructing models that can learn and adapt to market conditions. Once the need was clear, they were added and tested in less than four hours, and required no other changes in the rest of the system.
- MinneTAC is an open-source project, available at `http://tac.cs.umn.edu`. The source release includes the full infrastructure, and relatively simple examples of each of the decision components, a few evaluators, and a sample set of configuration files. It has been downloaded over 900 times since its initial release in April 2005. There have also been over 1000 binary downloads of the 2005 and 2006 competition versions of MinneTAC, which include some relatively complex evaluators that are not sufficiently documented or tested for source release. One user of the source download is a group at Kansas State University, whose agent MinneKatTac was entered in the 2007 competition. They have told us [16] that they evaluated multiple available source packages on which to build their agent, and MinneTAC was the easiest to understand and work with.

In any complex system, there are potential usability problems. A downside of configurability is the risk that the reduction in complexity on one area may be offset by an increase in complexity in another. The configuration needed to construct complex evaluator chains like the pricing chain shown in Figure 9 is large and difficult to understand in the verbose XML format. The version of MinneTAC used in the 2007 Trading Agent Competition uses 66 evaluators, with multiple feedback loops. This raises two questions: (1) Would any other approach lead to a system that is easier to understand? (2) Could a tool be found or built that would reduce the cognitive burden of constructing and understanding these complex configurations? We are in the process of constructing such a tool.

A second potential usability problem for complex software systems is the difficulty of determining whether they are actually working as intended. This problem is compounded by the TAC simulation environment, which imposes hard time limits on agent decision processes. This means that developers can-

[16] private communication

not stop and resume the agent with a debugging tool. But many systems operate with such constraints. A common solution is to write unit tests for individual components, and use standard program debugging tools in the test environment. This approach works well with the MinneTAC design.

7 Conclusions and Future Work

Future generations of e-commerce decision support systems will very likely involve a mixture of mixed-initiative and autonomous systems that will help organizations to discover opportunities, evaluate tradeoffs, coordinate processes within and across organizations, and conduct business in increasingly competitive markets. Developing the next generation of systems and understanding their dynamics will require a significant body of experimental work. The Trading Agent Competition is an example of an experimental environment that can stimulate needed developments.

Experimental work with multi-agent systems requires an implementation. Often, the design qualities that best support experimental work are different from those normally considered "ideal" in industry. In complex economic scenarios such as TAC SCM, 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 competitive environment, the ability to easily compose multiple agents with different combinations of decision process implementations makes it possible to test hypotheses about the effectiveness of competing decision models.

To understand how others have addressed the issues of agent design for the TAC SCM scenario, we conducted an informal survey of teams who have been involved in the competition over the last few years. Many of the top teams responded, and we see a wide variety of approaches. It appears that much of this variety arises from the broad range of research questions that are being addressed in the context of the competition. Many teams identified "modularity" as a significant feature of their designs. Indeed, modular software design appears to be a requirement for an agent that can remain viable in the competition environment for multiple years. This is because the research environment and the annual cycle of publication drives a need to frequently change implementation features in order to improve performance, test hypotheses, and gather data for empirical study.

We described the design of MinneTAC, an agent that is modular in the extreme. It is constructed using the Apache Excalibur component framework, which provides the ability to compose agent systems from sets of individual

components based on simple configuration files. We also showed that two basic mechanisms, event distribution with a variant of the Observer pattern, and composable Evaluations, permit an appropriate level of component interaction without introducing unnecessary coupling among components. The ability to compose complex evaluator chains out of relatively simple, straightforward elements has greatly simplified the design of the decision components themselves.

The MinneTAC architecture could clearly be used to implement a wide variety of agent behaviors, including those of most of our TAC SCM competitors. To do so, the problem of composing and understanding large evaluator networks must be solved. A graphical tool for visualizing and constructing evaluator chains is currently in development. A longer-term solution might be to add detailed semantic descriptions to the Evaluator interfaces, and enable semi-automatic composition. This is similar to the problem of composing Web Services, as described by Sirin et al. [33]. Rao et al. [32] argue that fully-automatic service composition may be an unrealistic goal, and propose a mixed-initiative approach.

The combination of Evaluators with Selectors and similar types of "switches" enables a range of behaviors that we have barely explored. 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.

Acknowledgments

We would like to thank the organizers of the TAC SCM game for a stimulating research problem and all the teams who have participated in our survey.

References

[1] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Addison Wesley, 1998.

[2] F. Bellifemine, A. Poggi, G. Rimassa, JADE–A FIPA-compliant agent framework, vol. 99, 1999.

[3] M. Benisch, J. Andrews, N. Sadeh, Pricing for customers with probabilistic valuations as a continuous knapsack problem, in: Proc. of 8th Int'l Conf. on Electronic Commerce, Fredericton, NB, 2006.

[4] M. Benisch, A. Greenwald, I. Grypari, R. Lederman, V. Naroditskiy, M. Tschantz, Botticelli: A supply chain management agent designed to optimize under uncertainty, ACM Trans. on Comp. Logic 4 (3) (2004) 29–37.

[5] M. Benisch, A. Sardinha, J. Andrews, N. Sadeh, CMieux: adaptive strategies for competitive supply chain trading, in: Proc. of 8th Int'l Conf. on Electronic Commerce, ACM Press, 2006.

[6] A. B. Binkley, S. R. Schach, A comparison of sixteen quality metrics for object-oriented design, Information Processing Letters 58 (1996) 271–275.

[7] F. Brito e Abreu, M. Goulão, R. Esteves, Toward the design quality evaluation of object-oriented software systems, in: Proceedings of the Fifth International Conference on Software Quality, Austin, Texas, 1995.

[8] D. Burke, K. Brown, S. Tarim, B. Hnich, Learning Market Prices for a Real-time Supply Chain Management Trading Agent, in: AAMAS06: Workshop on Trading Agent Design and Analysis (TADA/AMEC), 2006.

[9] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture: a System of Patterns, Wiley, 1996.

[10] B. Chaib-draa, J. Müller, Multiagent based Supply Chain Management (Studies in Computational Intelligence), Springer-Verlag New York, Inc. Secaucus, NJ, USA, 2006.

[11] K. C. Chatzidimitriou, A. L. Symeonidis, I. Kontogounis, P. A. Mitkas, Agent Mertacor: A robust design for dealing with uncertainty and variation in scm environments., Expert Systems with Applications (2007) Article in Press.

[12] S. Chopra, P. Meindl, Supply Chain Management, Pearon Prentice Hall, New Jersey, 2004.

[13] J. Collins, R. Arunachalam, N. Sadeh, J. Ericsson, N. Finne, S. Janson, The supply chain management game for the 2006 trading agent competition, Tech. Rep. CMU-ISRI-05-132, Carnegie Mellon University, Pittsburgh, PA (November 2005).

[14] A. S. Foundation, Apache excalibur, http://excalibur.apache.org/ (2006).

[15] D. Galindo, D. Ayala, F. L. Y. Lopez, Tiancalli06: An agent for the supply chain management game 2006, in: Proc. CIMCA/IAWTIC, IEEE Computer Society, 2006.

[16] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[17] R. Harrison, S. J. Counsell, R. V. Nithi, An evaluation of the MOOD set of object-oriented software metrics, IEEE Transactions on Software Engineering 24 (6) (1998) 491–496.

[18] M. He, A. Rogers, X. Luo, N. R. Jennings, Designing a successful trading agent for supply chain management, in: Proc. of the 5th Int'l Conf. on Autonomous Agents and Multi-Agent Systems, Hakodate, Japan, 2006.

[19] P. Jordan, C. Kiekintveld, M. Wellman, Empirical game-theoretic analysis of the TAC supply chain game, in: Proc. of the 6th Int'l Conf. on Autonomous Agents and Multi-Agent Systems, Hawaii, USA, 2007.

[20] M. Kapor, A software design manifesto, in: T. Winograd (ed.), Bringing Design to Software, ACM Press, 1996, pp. 1–9.

[21] P. Keller, F.-O. Duguay, D. Precup, Redagent-2003: An autonomous market-based supply-chain management agent, in: Proc. of the Third Int'l Conf. on Autonomous Agents and Multi-Agent Systems, ACM, ACM Press, New York, NY, USA, 2004.

[22] W. Ketter, J. Collins, M. Gini, A. Gupta, P. Schrater, Identifying and forecasting economic regimes in TAC SCM, in: H. L. Poutré, N. Sadeh, S. Janson (eds.), Agent-Mediated Electronic Commerce: Designing Trading Agents and Mechanisms, vol. 3937 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 2006, pp. 113–125.

[23] W. Ketter, J. Collins, M. Gini, A. Gupta, P. Schrater, A predictive empirical model for pricing and resource allocation decisions, in: Proc. of 9th Int'l Conf. on Electronic Commerce, Minneapolis, Minnesota, USA, 2007.

[24] W. Ketter, J. Collins, M. Gini, A. Gupta, P. Schrater, Detecting and Forecasting Economic Regimes in Multi-Agent Automated Exchanges, Decision Support Systems Forthcoming - accepted for publication (2008) –.

[25] W. Ketter, E. Kryzhnyaya, S. Damer, C. McMillen, A. Agovic, J. Collins, M. Gini, Analysis and design of supply-driven strategies in TAC-SCM, in: AAMAS04: Workshop on Trading Agent Design and Analysis, New York, 2004.

[26] C. Kiekintveld, J. Miller, P. Jordan, M. P. Wellman, Controlling a supply chain agent using value-based decomposition, in: Proc. of 7th ACM Conf. on Electronic Commerce, Ann Arbor, Mich., 2006.

[27] C. Kiekintveld, M. P. Wellman, S. Singh, J. Estelle, Y. Vorobeychik, V. Soni, M. Rudary, Distributed feedback control for decision making on supply chains, in: Proc. Int'l Conf. on Automated Planning and Scheduling, Whistler, BC, Canada, 2004.

[28] I. Kontogounis, K. C. Chatzidimitriou, A. L. Symeonidis, P. A. Mitkas, A Robust Agent Design for Dynamic SCM environments, in: 4th Hellenic Conference on Artificial Intelligence (SETN'06), Heraklion, Crete, Greece, 2006.

[29] R. Martin, Object-oriented design quality metrics: an analysis of dependencies, Report on object analysis and design 2.

[30] D. Pardoe, P. Stone, Tactex-05: A champion supply chain management agent, in: Proc. of the Twenty-First Nat'l Conf. on Artificial Intelligence, AAAI, Boston, Mass., 2006.

[31] V. Podobnik, A. Petric, G. Jezic, The CrocodileAgent: Research for Efficient Agent-Based Cross-Enterprise Processes, Lecture Notes in Computer Science 4277 (2006) 752–762.

[32] J. Rao, D. Dimitrov, P. Hofmann, N. Sadeh, A mixed-initiative semantic web framework for process composition, in: Proc. of the 5th International Semantic Web Conference, Springer, Athens, GA, USA, 2006.

[33] E. Sirin, J. Hendler, B. Parsia, Semi-automatic composition of web services using semantic descriptions, in: Web services: modeling, architecture and infrastructure workshop, ICEIS 2003, Angers, France, 2003.

[34] E. Sodomka, J. Collins, M. Gini, Efficient statistical methods for evaluating trading agent performance, in: Proc. of the 22nd Nat'l Conf. on Artificial Intelligence, AAAI, AAAI Press, Vancouver, BC, 2007.

[35] C. Szyperski, Component Software: Beyond Object-Oriented Programming, ACM Press, 1998.

[36] P. Vytelingum, R. Dash, M. He, N. Jennings, A Framework for Designing Strategies for Trading Agents, Proc. IJCAI Workshop on Trading Agent Design and Analysis (2005) 7–13.

[37] M. P. Wellman, J. Estelle, S. Singh, Y. Vorobeychik, C. Kiekintveld, V. Soni, Strategic interactions in a supply chain game, Computational Intelligence 21 (1) (2005) 1–26.

[38] R. J. Wieringa, Design Methods for Reactive Systems, Morgan Kaufmann, San Francisco, 2003.

[39] I. H. Witten, E. Frank, Data Mining: Practical Machine Learning Tools and Techniques, Second Edition, Morgan Kaufmann, 2005.

[40] M. Wooldridge, Introduction to Multiagent Systems, John Wiley & Sons, Inc. New York, NY, USA, 2001.

Appendix: TAC SCM Design Questionnaire

The following questions were used in the design survey described in Section 3:

(1) Which team do you represent? What has been your role on the team?
(2) What are the main goals of your design apart from winning the game?
(3) What are your organizing design principles (architectural style, major modules and responsibilities)?
(4) What are the strengths and weaknesses of your design? In other words, what is easy and what is hard to do given your design? To what extent do you feel your design has met your goals?
(5) If you have been in the competition for more than two years, have you made significant changes in your design and why?
(6) Does your design represent a significant departure from the Agentware package?

(7) Which significant 3rd party packages have you used, e.g. Weka, CPlex, Apache Excalibur, Jade, etc.?
(8) Have you based your design on a publicly-available agent design, like TacTex, GeminiJK, or MinneTAC?
(9) Have you published information about your agent design? If yes, where?