Architectures for agents in TAC SCM

John Collins
Dept of CSE
University of Minnesota
Minneapolis, MN

Wolfgang Ketter
Dept of DIS
Erasmus University
Rotterdam, NL

Maria Gini[*]
Dept of CSE
University of Minnesota
Minneapolis, MN

Abstract

An autonomous trading agent is a complex piece of software that must operate in a competitive economic environment. We report results of an informal survey of agent design approaches among the competitors in the Trading Agent Competition for Supply Chain Management (TAC SCM).

Introduction

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 provide an overview of the design choices made by the designers of agents for the Trading Agent Competition for Supply-Chain Management (Collins *et al.* 2005) (TAC SCM), and to describe the design of the MinneTAC trading agent, which has competed effectively in TAC SCM for several years.

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. Agents compete with each other in a procurement market for computer components, and in a sales market for customers. A game runs for 220 simulated days over about an hour of real time. Each agent starts with no inventory and an empty bank account. The agent with the largest bank account at the end of the game is the winner.

Customers express demand each day by issuing a set of 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.

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. Many factors, such as current capacity, outstanding commitments of suppliers, sales volumes and price distribution in the customer market, are not visible to the agents.

Team	University	Team contact and role
Botticelli (B)	Brown University (USA)	Victor Naroditskiy (team member for 4 years)
CMieux (CM)	Carnegie Mellon University (USA)	Michael Benisch (team leader 2005/2006)
CrocodileAgent (CA)	University of Zagreb (Croatia)	Vedran Podobnik (team leader)
DeepMaize (DM)	University of Michigan (USA)	Chris Kiekintveld (team member for 5 years)
Foreseer (F)	Cork Constraint Computation Centre (Ireland)	David Burke (main developer)
Mertacor (M)	Aristotle University of Thessaloniki (Greece)	Andreas Symeonidis (team member 2006)
MinneTAC (MT)	University of Minnesota (USA)	John Collins (team leader and designer)
Southampton (S)	University of Southampton (UK)	Minghua He (designer and programmer)
TacTex (TT)	University of Texas (USA)	David Pardoe (main developer for 5 years)
Tiancalli (T)	Benemerita Universidad Autonoma de Puebla (Mexico)	Daniel Macas Galindo (team leader)

Table 1: Participating teams in the TAC SCM architecture questionnaire.

Designs of agents for TAC SCM

We report findings from an informal survey which 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 the best teams in previous tournaments. In Table 1 we list the teams who responded to the questionnaire. We categorize 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 (see Table 2).

Constraint optimization. A supply-chain trading agent must make its decisions subject to a number of internal and external constraints. These constraints apply to parts of the supply-chain, such as procurement (availability of supplies, reputation), production (limited production capacity), sales (limited demand, variable pricing), and fulfillment (shipments limited by finished goods inventory). Nearly all the teams who answered our questionnaire used some form of constraint optimization, so we list here the ones who highlighted it. The teams who focus on realtime optimization, Botticelli (Benisch *et al.* 2004), DeepMaize (Kiekintveld *et al.* 2006), Foreseer (Burke *et al.* 2006), MinneTAC (Ketter *et al.* 2007) use mainly third party optimization packages, including CPLEX (http://www.ilog.com/products/cplex/), and lp_solve (http://sourceforge.net/projects/lpsolve). An exception is CMieux (Benisch *et al.* 2006) which uses an internally-developed algorithm to solve a continuous knapsack problem for pricing customer offers.

Machine learning. Many agents use machine learning algorithms to learn from historical market data and have some ability to learn during operation to adapt to changing situations. CMieux (Benisch *et al.* 2006), MinneTAC (Ketter *et al.* 2007), and TacTex (Pardoe & Stone 2006) identified the need to support learning and adaptation as primary concerns in the design of their agents. Both CMieux (Benisch *et al.* 2006) and TacTex (Pardoe & Stone 2006) use the Weka (http://www.cs.waikato.ac.nz/ml/weka/) machine learning tool set. MinneTAC is using Matlab (http://www.mathworks.com/) in combination with the Netlab (http://www.ncrg.aston.ac.uk/netlab/) 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.

Management of dynamic supply chains. The TAC SCM simulation is an abstract model of a highly dynamic direct sales environment. Many teams have a strong research focus on dynamic supply-chain behavior. These include CMieux (Benisch *et al.* 2006), Foreseer (Burke *et al.* 2006), Mertacor (Kontogounnis *et al.* 2006), MinneTAC (Ketter *et al.* 2007), and Tiancalli (Galindo, Ayala, & Lopez 2006). As a consequence teams strive for high flexibility in their agent design, so that they can easily accommodate changes.

Telecommunication. The CrocodileAgent (Podobnik, Petric, & Jezic 2006) group is part of a larger group that focuses on autonomous agents for management of large-scale telecommunication networks. They view TAC SCM as a challenge in building an agent that can operate in a dynamic environment, but they are

Research Agenda	Team	Architectural Emphasis
Constraint optimization	B, CM, DM, F, MT	3rd party packages
Machine learning	CM, MT, TT	External analysis framework, 3rd party packages
Dynamic supply-chain	CM, F, M, MT, T	Flexibility
Telecommunication	CA	Scalability
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, MT, S	Modularity
Dealing with uncertainty	B, S	Modularity

Table 2: Research agendas of teams and their architectural emphasis.

also concerned with scalability and other issues that go far beyond TAC SCM. They base their design on the JADE (`http://jade.tilab.com/`) framework, which is well-proven for large-scale multi-agent systems.

Architecture. CrocodileAgent (Podobnik, Petric, & Jezic 2006) and Southampton SCM (He *et al.* 2006) have structured their agent decision processes around the the IKB (Information, Knowledge, and Behavior) model (Vytelingum *et al.* 2005), a three layered agent-based framework. The first layer contains data gathered from the environment, the second knowledge extracted from the data, and the third encapsulates the reasoning and decision-making components that drive agent behavior. CrocodileAgent (Podobnik, Petric, & Jezic 2006) has also adopted the IKB approach. An advantage of using JADE is that the separation of I, K & B layers enables physical distribution of layers on multiple computers. Information layer agents parse out information from the TAC SCM server messages, while information and knowledge flows are implemented as JADE agent communication-based messages.

MinneTAC uses a component based framework. All data to be shared among components are kept in the Repository, which acts as a blackboard (Buschmann *et al.* 1996). MinneTAC is the only team that emphasized a design that attempts to minimize the learning curve for a researcher who wishes to work on a specific subproblem.

Empirical game theory. The DeepMaize (Jordan, Kiekintveld, & Wellman 2007) team pursues empirical game-theoretic analysis as one of their major research cornerstones. They employ an experimental 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 (Jordan, Kiekintveld, & Wellman 2007). 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.

Decision coordination. The supply-chain scenario places a premium on effective coordination of decisions affecting multiple markets and internal resources. Decision coordination is an explicit emphasis for the Deep-Maize (Kiekintveld *et al.* 2006), MinneTAC (Ketter *et al.* 2007), and Southampton (He *et al.* 2006) teams. This problem is commonly viewed as one of enabling independent decision processes to coordinate their actions while minimizing the need to share representation and implementation details.

MinneTAC uses a blackboard approach to allow decision processes to coordinate their actions through a common state representation. Southampton uses the hierarchical IKB approach, in which the Knowledge layer can be viewed as a type of blackboard. Deep-Maize (Kiekintveld *et al.* 2006) 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.

Dealing with uncertainty. TAC SCM is designed to force agents to deal with uncertainty in many dimensions. Sodomka et al. (Sodomka, Collins, & Gini 2007) provide a good overview of the sources of uncertainty in the context of an empirical study of agent performance. The Botticelli group clearly identifies the problem of dealing with uncertainty as one of their main research goals in (Benisch *et al.* 2004). SouthamptonSCM (He *et al.* 2006) employs a bidding strategy that uses fuzzy logic to adapt prices according to the uncertain market situation.

The design of MinneTAC

In designing MinneTAC we follow a component-oriented approach. The idea is to provide an infrastructure that manages data and interactions with the game server, and cleanly separates behavioral components from each other. This allows 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 Apache Excalibur component framework (`http://excalibur.apache.org/`) 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.

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. All data that must be shared among components is kept in the Repository, which acts as a blackboard (Buschmann *et al.* 1996). 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.
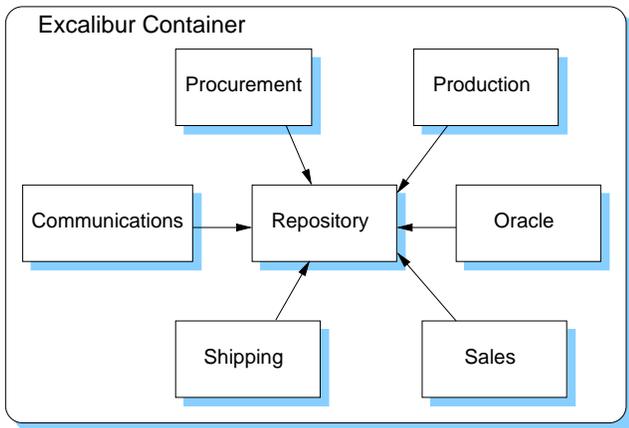

Figure 1: MinneTAC Architecture. Arrows indicate dependencies.

Repository.

The Repository is the one component that is visible to other components. Its primary functions are storage and management of agent state, event distribution, and support of the Evaluation mechanism discussed below. At the beginning of each day of a game, new incoming messages are received from the game server and deposited into the Repository. Once the full set of incoming messages has been received, events are generated to notify other components to perform their analyses and decisions. In response to these events, the decision components retrieve data and evaluations from the Repository, and record their decisions back into it. Finally, the resulting decisions are retrieved from the Repository by the Communications component and returned

to the server.

From an architectural standpoint, the Repository plays the part of the Blackboard in the *Blackboard* pattern (Buschmann *et al.* 1996), and the remainder of the components, other than the Communications component, act as Knowledge Sources. The Control element of the Blackboard pattern is replaced by the Event and the Evaluable/Evaluator mechanisms, which we now describe.

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.

Events are generated in response to state transitions, as visualized in Figure 2. Each simulated day begins in the *start of day* state, during which the agent waits for the first message from the server. In the *receiving* state, MinneTAC handles all incoming data messages by storing them in the Repository. Completion of the daily batch is signified by receipt of a "sim-status" message. The Repository responds with a transition to the *evaluating* state and generation of the *data-available* event. When a component receives *data-available* it is able to inspect the incoming data for the day's transactions and perform whatever analysis is needed to update its models. When this is complete, the Repository generates the *decision* event, and transitions to the *deciding* state. When a component receives the *decision* event, it is expected to finalize its decisions and record its outgoing messages back in the Repository. At the conclusion of decision processing, the accumulated messages are returned to the game server, and the Repository returns to the *start of day* state.
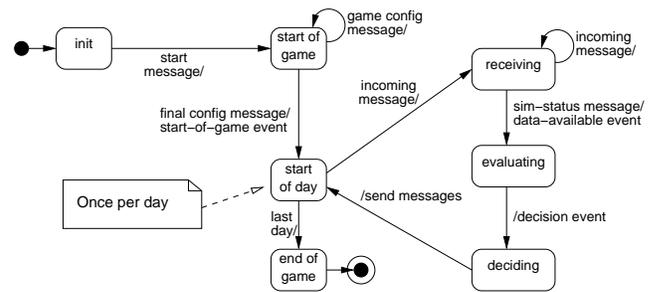

Figure 2: States and transitions in the Repository component. Arcs are labeled with event/action pairs.

In processing the *data-available* and *decision* events, the Repository acts as Subject and the other components as Observers in the *Observer* pattern (Gamma *et al.* 1995). This approach has the advantage of eliminating dependency of the Repository on the specific components. However, an important limitation of the

Observer pattern is that the sequence of notifications is not controlled, although in most (single-threaded) 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 a configuration file, not the code.

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, instances of the type ConfiguredObject or its subtypes. These are used to implement market models and to perform analysis and prediction tasks. The principal subtype is Evaluator, but a few other types are supported as well. The Oracle itself reads its configuration data when the agent starts, and uses it to create and configure the specified objects. ConfiguredObject is an abstract class that has a name and an ability to configure itself, given an XML clause. The Oracle creates ConfiguredObject instances and keeps track of them by mapping their names to the created instances.

Another interesting subtype of ConfiguredObject is the Selector. A Selector is 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. The behavior of the market during this early period is fairly predictable. Later in the game, prices are less predictable and more sophisticated models may be useful. A DateSelector can be used to switch between pricing models at a particular preset dates, or a more sophisticated Selector can be used to switch models once the initial price decline bottoms out.

Evaluators. To minimize coupling between components when a component needs to make a decision, it will inspect its own internal data and data in the Repository and run some function. Operations on Repository data can be encapsulated in the form of Evaluations,

and made available to other components.

All the major data elements in the Repository are Evaluable types. Each Evaluable can be associated with some number of associated Evaluations and with 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. Evaluators implement back-chaining by requesting other Evaluations in the process of producing their results. 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. Figure 3 shows an 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 would need parts cost information and *price*. The *sort-by-profit* evaluation would need the *profit* evaluations on the individual RFQs.
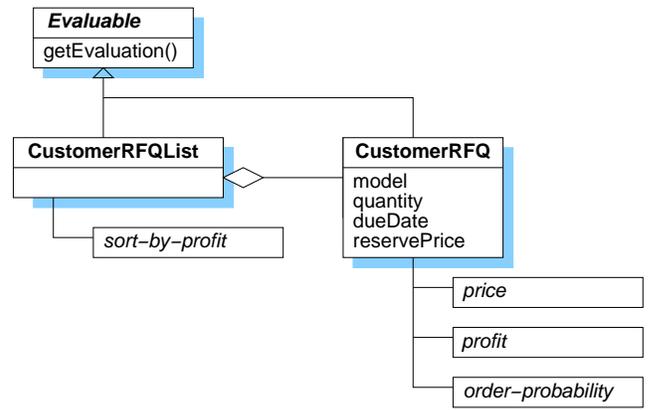

Figure 3: RFQ evaluation example.

Example of using Evaluators for Sales

To illustrate the power of Evaluators, we show in Figure 4 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 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 for each product type is the difference between Evaluations called *median-price* and *cost-basis* for those products.
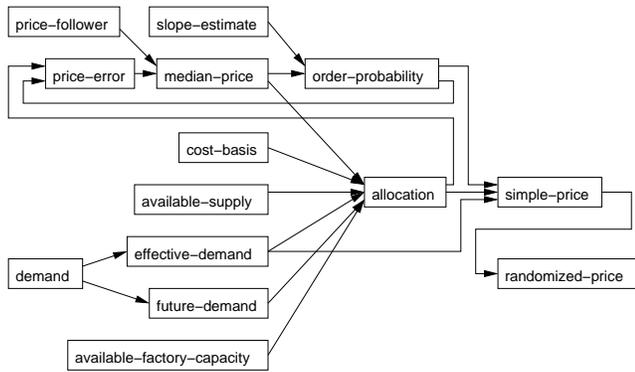

Figure 4: Evaluator chain for sales quota and pricing.

The Evaluation generated by the *allocation* evaluator gives desired sales quotas for each product over a time horizon. Given a sales quota for a given 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.

Conclusions and Future Work

The survey outcome shows that there are common themes emerging from the different research groups on how to design a successful agent architecture. in addition to problem-specific approaches. 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.

With the design of MinneTAC 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.

References

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

Benisch, M.; Sardinha, A.; Andrews, J.; and Sadeh, N. 2006. CMieux: adaptive strategies for competitive supply chain trading. In *Proc. of 8th Int'l Conf. on Electronic Commerce*, 47–58. ACM Press.

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

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

Collins, J.; Arunachalam, R.; Sadeh, N.; Ericsson, J.; Finne, N.; and Janson, S. 2005. The supply chain management game for the 2006 trading agent competition. Technical Report CMU-ISRI-05-132, Carnegie Mellon University, Pittsburgh, PA.

Galindo, D.; Ayala, D.; and Lopez, F. L. Y. 2006. Statistic analysis for the Tiancalli agents on TAC SCM 2005 and 2006. In *Proc. 15th Int'l Conf. on Computing*, 161–166.

Gamma, E.; Helm, R.; Johnson, R.; and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Professional Computing Series. Addison-Wesley.

He, M.; Rogers, A.; Luo, X.; and Jennings, N. R. 2006. Designing a successful trading agent for supply chain management. In *Proc. of the 5th Int'l Conf. on Autonomous Agents and Multi-Agent Systems*.

Jordan, P.; Kiekintveld, C.; and Wellman, M. 2007. 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*.

Ketter, W.; Collins, J.; Gini, M.; Gupta, A.; and Schrater, P. 2007. A predictive empirical model for pricing and resource allocation decisions. In *Proc. of 9th Int'l Conf. on Electronic Commerce*, 449–458.

Kiekintveld, C.; Miller, J.; Jordan, P.; and Wellman, M. P. 2006. Controlling a supply chain agent using value-based decomposition. In *Proc. of 7th ACM Conf. on Electronic Commerce*, 208–217.

Kontogounnis, I.; Chatzidimitriou, K. C.; Symeonidis, A. L.; and Mitkas, P. A. 2006. A Robust Agent Design for Dynamic SCM environments. In *4th Hellenic Conference on Artificial Intelligence (SETN'06)*, 127–136.

Pardoe, D., and Stone, P. 2006. Tactex-05: A champion supply chain management agent. In *Proc. of the Twenty-First Nat'l Conf. on Artificial Intelligence*, 1389–1394. Boston, Mass.: AAAI.

Podobnik, V.; Petric, A.; and Jezic, G. 2006. The CrocodileAgent: Research for Efficient Agent-Based Cross-Enterprise Processes. *Lecture Notes in Computer Science* 4277:752–762.

Sodomka, E.; Collins, J.; and Gini, M. 2007. Efficient statistical methods for evaluating trading agent

performance. In *Proc. of the 22nd Nat'l Conf. on Artificial Intelligence*, 770–775. Vancouver, BC: AAAI.

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