Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 04-017

Component-based design for a trading agent

John Collins, Amrudin Agovic, Steven Damer, and Maria Gini

May 04, 2004

Component-based design for a trading agent

John Collins, Amrudin Agovic, Steven Damer, and Maria Gini

Dept. of Computer Science and Engineering,
University of Minnesota

Abstract. MinneTAC is an agent designed to compete in the Supply-Chain Trading Agent Competition [1]. It is also designed to support the needs of a group of researchers, each of whom is interested in different decision problems related to the competition scenario. The design of MinneTAC breaks out each basic behavior into a separate, configurable component. Dependencies between components are almost non-existent. The agent itself is defined as a set of "roles", and a working agent is one for which a component is supplied for each role. This allows each user to focus on a single problem and work independently, and it allows multiple user to tackle the same problem in different ways. A working MinneTAC agent is completely defined by a set of configuration files that map the desired roles to the code that implements them, and that set parameters for the components. This paper describes the design of MinneTAC and evaluates its effectiveness in support of our research agenda, and in its competitiveness in the TAC-SCM game environment.

1 Introduction

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

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

This paper describes the design of an agent for TAC SCM. 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 both in terms of the competitiveness of the agents that have been implemented with it, and in terms of its ability to support our research agenda.

2 Overview of the TAC SCM game

In a TAC SCM game, each of the competing agents plays the part of a manufacturer of personal computers. Agents compete with each other in a procurement market for computer components, and in a sales market for customers, as shown in Figure 1. 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 an initial parts inventory before it can begin assembling and shipping computers. The agent with the largest bank account at the end of the game is the winner.

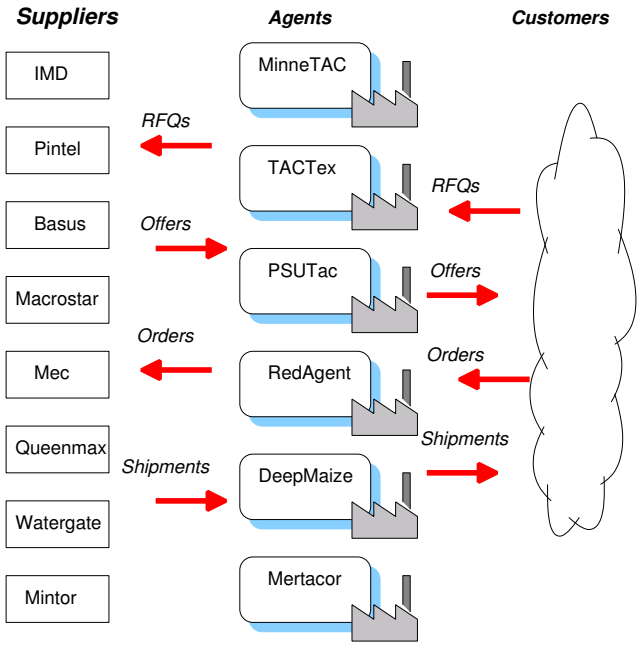

Fig. 1. Schematic overview of a typical TAC SCM game scenario.

2.1 Game scenario

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. Each agent may choose to bid on some or all of the day's RFQs. For each RFQ, the one with the lowest price will be accepted, as long as that price is at or below the reserve price. Once a bid is accepted, the agent is obligated to ship the requested products by the due date, or it pays 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 issues 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 at or after the due date (supplier capacity is variable). Suppliers do not pay penalties for late shipments.

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

2.2 Agent decisions

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

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

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

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

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

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

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

3 Design challenges

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

For example, our design must support multiple independent developers pursuing their own lines of research. The TAC SCM scenario presents a number of relatively independent decision problems, and there are many possible approaches to solving them. Our design must make it relatively easy for a researcher

to focus on a particular subproblem without having to worry about getting a whole agent to work correctly. We also need to be able to configure agents with different combinations of decision process implementations.

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

Decision processes may involve somewhat arbitrary parameters, and their interactions and the sensitivity of agent performance to the settings of those parameters may not be well-defined. This is true even in cases where the agent is designed specifically to minimize the number of such parameters by use of optimization methods [8].

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.

4 The design of MinneTAC

To address the design challenges of the MinneTAC agent, we follow a component-oriented approach [14]. The idea is to provide an infrastructure that manages data and interactions with the game server, allowing individual researchers to encapsulate agent decision problems within the bounds of individual components that have minimal dependencies among themselves. Two pieces of software form the foundation of MinneTAC: the Avalon component framework, and the "dummy agent" distributed by the TAC SCM game organizers. Avalon provides the standards and tools to build components and configure working agents from collections of individual components, and the dummy agent handles interaction with the game server.

4.1 A brief overview of Avalon

Apache Avalon [9] is a general-purpose component framework. It is widely used, primarily as a foundation for middleware and server sofware, such as the OpenORB CORBA implementation (OpenORB.sourceforge.net) but its use in the implementation of autonomous agents is rare. It does not provide the "classic" facilities for agent design, such as knowledge representation, inter-agent communication, reasoning facilities, or a planning infrastructure. Instead, it provides a means to build complex, robust systems from sets of role-based, configurable components. This satisfies a primary goal of MinneTAC, allowing researchers to work

independently on individual decision problems with minimal need for detailed coordination with each other.

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

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

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

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

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

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

Startable. The component runs a thread of control.

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

4.2 MinneTAC architecture

Figure 2 is a high-level view of the architecture of MinneTAC. It is conceptually simple: The Avalon container, and (at least) 7 components. All data that must be shared among components is kept in the Repository, which acts as a blackboard [4]. The components themselves are identified by their roles; in several cases multiple components have been built to fill those roles. It is an explicit goal of this architecture to minimize couplings between the components. Ideally, each component depends only on Avalon and the Repository.

The MinneTAC agent opens three configuration files when it starts. These are handed to the Avalon infrastructure, which reads and processes them. The

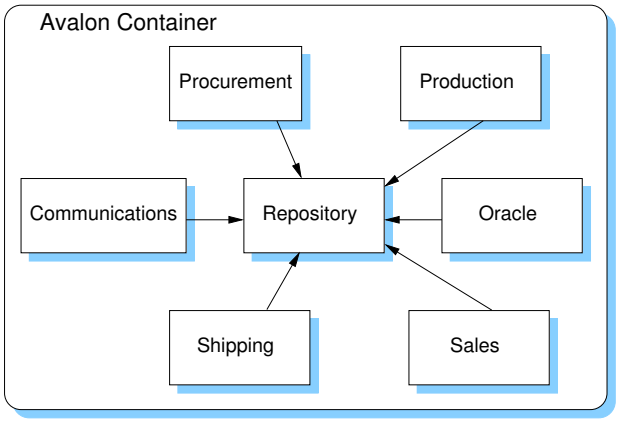

Fig. 2. MinneTAC Architecure. Arrows indicate API dependencies.

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

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

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

As shown in Figure 3, the agent does not need to react to individual messages from the server. Instead, it waits until after all the day's messages have been received, and then considers all of them together. In fact, there is one additional

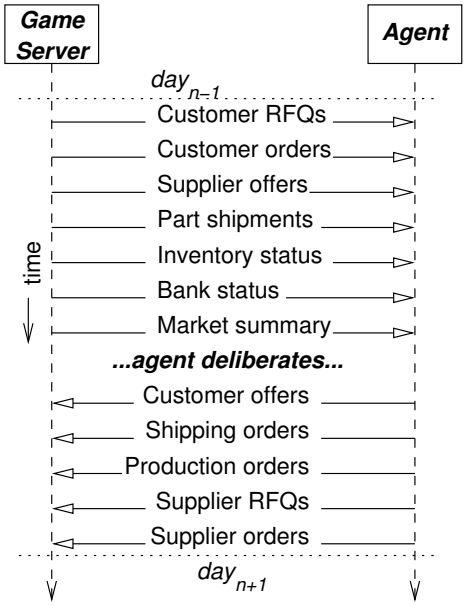

Fig. 3. One day of communications activity between the game server and an agent.

end-of-data message not shown in the figure, which contains no data but simply tells the agent that the day's input messages are complete. MinneTAC handles all data messages by storing them in the Repository. When the end-of-data message is handled by the Repository, it notifies the other components that the day's data input is complete. Components use this notification as the signal to perform their deliberations and compose the day's return messages. In this interaction, the Repository acts as a Subject and the other components as Observers in the *Observer* pattern [5].

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

Evaluators. As we stated in Section 4.2, a goal of the MinneTAC design is to minimize coupling between the various components. How, then, do they communicate? One possible approach is the one used by the RedAgent team at McGill, in which the components communicate through internal auction-based markets. Our approach is to use *evaluations* that are attached to the various data elements in the Repository. The general idea is that when a component needs to make a decision, it will inspect the available data and run some utility-maximizing function. The available data consists of any data it maintains internally, and the data in the repository. Any data reductions or analyses that are performed on

Repository data can be encapsulated in the form of Evaluations, and added back to the repository, where they will be visible to the other components.

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

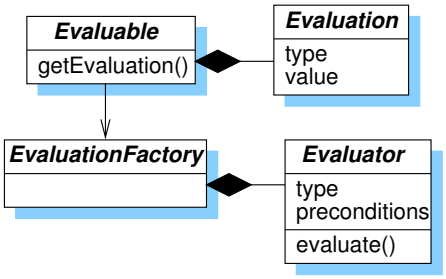

Fig. 4. Evaluables, Evaluations, and Evaluators.

Figure 5 shows a simple example of some Evaluable instances and a set of Evaluations that might be associated with them. The *price* evaluation could be supplied by the Sales component, and it might require parts cost information from the Procurement component as well as an estimate of current market conditions from the Oracle component. The *profit* evaluation would need parts cost information and *price*. The *sort-by-profit* evaluation would need the *profit* evaluations on the individual RFQs.

4.3 MinneTAC components

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

Repository is the unifying element of the MinneTAC design, the one component that is visible to the other components. It serves as an internal database,

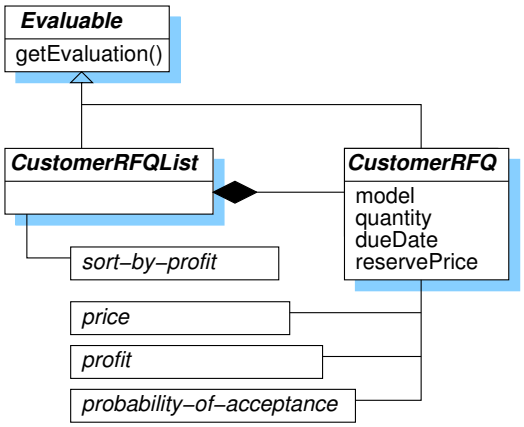

Fig. 5. RFQ evaluation example.

maintains the state of the system, and notifies other components of changes in state. All other activity is driven by these state changes. Ii also provides the core elements of the Evaluation subsystem.

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

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

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

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

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

Oracle predicts future demand and availability.

Repository. As stated earlier, the Repository is the one component that is visible to all the other components. The Communications component deposits new incoming messages into the Repository at the beginning of a day, waits for

the decision processes to complete, and retrieves the agent's responses from the Repository at the end of the day. Other components are notified when they must make their decisions. To perform their evaluations, they retrieve data from the repository, and record their decisions in the repository.

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

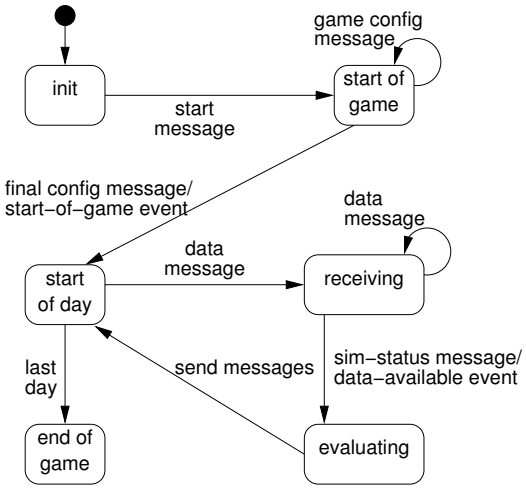

Fig. 6. States and transitions in the Repository component.

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

The Repository plays the part of the Blackboard in the *Blackboard* pattern [4], and the remainder of the components, other than the Communications component, act as Knowledge Sources. However, the Control element of the Blackboard pattern is replaced by the Evaluable/Evaluator mechanism.

Communications. In order to participate in a trading game, the agent must be able to communicate with the game server. The basic communication behaviors are provided in a *dummy agent*, provided by the game organizers. One way to

construct an agent for the TAC game is to use the communication elements in the dummy agent code directly. However, our team felt that this was a risky approach, since changes to the communication protocol could ripple through to changes in the dummy agent, necessitating new rounds of code-extraction and disruptions to our own existing code. Our approach is therefore to simply "wrap" the entire dummy agent with an Avalon component that has responsibility for communications with the game server. The Communications component acts as an *Adapter* in the sense of [5]. We show this approach schematically in Figure 7. This way, we avoid modifying the dummy agent, and we are always able to use the latest version of the dummy agent by downloading it and importing it into our environment.

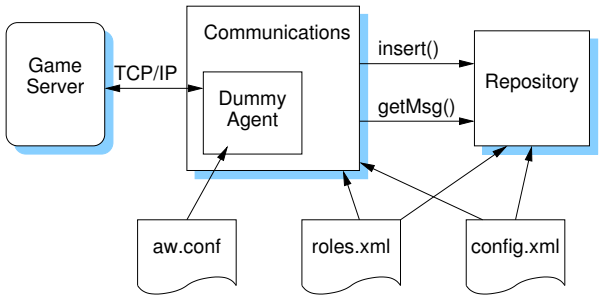

Fig. 7. Communications component wraps the Dummy Agent.

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

5 Evaluation

We evaluate the success of the MinneTAC design by asking two questions: (1) Does the agent perform well, and to what extent does the design affect agent performance? (2) Does the design meet the "usability" challenges described in Section 3?

5.1 Performance

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

The Avalon framework does indeed impose some overhead when the agent starts up, since it must read configuration files, find and load code for components, and set up and configure the components. However, once the agent is

running, there is essentially no overhead imposed by the framework. Event processing and evaluation is done by direct lookup, since event handlers and Evaluators are registered when components are loaded. We have run 6 MinneTAC agents on the same desktop machine (a 1 GHz Pentium), and all 6 agents are able to complete their daily decision procedures in less than 1 second.

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

5.2 Usability

Mitch Kapor says that software design bridges the world of people and human purpose with the world of technology [6]. It's easy to understand what that means when the artifact is a desktop application intended to be used directly by people. But in fact, the first user of a software design is the programmer who implements it. From the implementer's standpoint, good design is easy to understand, easy to implement correctly, and easy to maintain.

MinneTAC is an autonomous agent. It exists in the game environment and has no user interface, other than the configuration files it reads and the logfiles it produces. The principal usability criterion is whether researchers can effectively work on the various decision problems independently, and whether they can extract the data they need to analyze performance and confirm or refute hypotheses. Inexperienced student programmers have been able to contribute significant functionality, and a wide variety of analyses have been carried out. An example of these analyses is in [7].

6 Related Work

Most agent design efforts have focused on either the autonomous behavior aspects of agency, or on interaction among agents. Shoham's Agent-Oriented Programming [12] examines a cognitive and societal view of computation. Bradshaw's KAoS agents [2] are BDI agents in a CORBA environment. Agents have *capabilities* based on existing document management applications. Norman *et al.* [11] describe agent societies that model organizational structures and automate business processes. These ADEPT agents negotiate over service agreements that can involve many parties and many dimensions. JADE [10] is an agent framework that has been used to build trading agents, and could have been used for MinneTAC. However, its primary emphasis is on building multi-agent systems that comply with FIPA specifications for inter-agent communications, and with flexible deployment in a network environment. This is not a requirement for the TAC SCM domain. The MinneTAC design is *compositional* in the sense

of Brazier *et al.* [3], but not hierarchically so. The DESIRE method from Brazier *et al.* does not seem applicable to the MinneTAC situation, since we are dealing with a single agent in an existing environment, and the blackboard approach used in MinneTAC is not easily modeled with DESIRE. RETSINA [13] suggests both a multi-agent architecture with a variety of agent roles, and an architecture for individual agents that provides communications, planning, scheduling, and execution monitoring. This architecture could probably be adapted to the TAC SCM domain, but its planning and communication capabilities would not be especially useful. Vetsikas and Selman [15] show a method for studying design tradeoffs in a trading agent. This approach could be likely be used effectively in MinneTAC.

Ultimately, the TAC SCM problem domain does not require the sort of flexible cognitive and social elements of these more "traditional" agent designs. Instead, our focus has been on separating the decision tasks and supporting research needs, and we have found the component-oriented model to be ideal.

7 Conclusions and Future Work

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

We show one way to construct such an agent, using a readily-available component framework. The framework provides the ability to compose agent systems from sets of individual components based on simple configuration files. We also show that two basic mechanisms, event distribution with the Observer pattern and our Evaluator-Evaluation scheme, permit an appropriate level of component interaction without introducing unnecessary coupling among components.

There are many possible extensions to the basic design we show here. One that we are currently pursuing is to add an "executive" component that would allocate "resources" to competing implementations of basic decision processes within a single agent. This would allow a high degree of adaptability in the game environment, where the level of demand can fluctuate greatly, and where the actions of other agents can have a significant impact on the markets.

8 Acknowledgement

We would like to thank the organizers of the TAC SCM game for a stimulating research problem. Partial support for this research is gratefully acknowledged from the National Science Foundation under award NSF/IIS-0084202.

References

1. Raghu Arunachalam, Joakim Eriksson, Niclas Finne, Sverker Janson, and Norman Sadeh. The TAC supply chain management game. Technical Report Draft Version 0.62, Swedish Institute of Computer Science, Sweden, 2003.
2. Jeffrey M. Bradshaw, Stewart Dutfield, Pete Benoit, and John D. Wolley. KAoS: Toward an industrial-strength open agent architecture. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 375–418. AAAI Press, 1997.
3. Frances M. T. Brazier, Catholign M. Jonker, and Jan Treur. Principles of component-based design of intelligent agents. *Data and Knowledge Engineering*, 41(1):1–28, April 2002.
4. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: a System of Patterns*. Wiley, 1996.
5. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.
6. Mitchell Kapor. A software design manifesto. In Terry Winograd, editor, *Bringing Design to Software*, pages 1–9. ACM Press, 1996.
7. Wolfgang Ketter, Elena Kryzhnyaya, Steven Damer, Colin McMillen, Amrudin Agovic, John Collins, and Maria Gini. MinneTAC sales strategies for supply chain TAC. In *Proc. of the Third Int'l Conf. on Autonomous Agents and Multi-Agent Systems*, page submitted, New York, July 2004.
8. C. Kiekintveld, M. P. Wellman, S. Sing, J. Estelle, Y. Vorobeychik, V. Soni, and M. Rudary. Distributed feedback control for decision making on supply chains. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling*, 2004.
9. B. Loritsch. Developing with Apache Avalon. Apache Software Foundation, 2001.
10. P. Moraitis, E. Petraki, and N. Spanoudakis. Engineering JADE agents with the Gaia methodology. In R. Kowalszyk, J. Miller, H. Tianfield, and R. Unland, editors, *Agent Technologies, Infrastructures, Tools, and Applications for e-Services*, volume 2592 of *Lecture Notes in Computer Science*, pages 77–91. Springer-Verlag, 2003.
11. Timothy J. Norman, Nicholas R. Jennings, Peyman Faratin, and E. H. Mamdani. Designing and implementing a multi-agent architecture for business process management. In M. J. Wooldridge, J. P. Müller, and N. R. Jennings, editors, *Intelligent Agents III*, volume 1193 of *Lecture Notes in Artificial Intelligence*, pages 261–275. Springer-Verlag, Berlin, 1997.
12. Yoav Shoham. An overview of agent-oriented programming. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 271–290. AAAI Press, 1997.
13. Katia Sycara and Anandeep S. Pannu. The RETSINA multiagent system: towards integrating planning, execution, and information gathering. In *Proc. of the Second Int'l Conf. on Autonomous Agents*, pages 350–351, 1998.
14. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, 1998.

15. Joannis A. Vetsikas and Bart Selman. A principled study of the design tradeoffs for autonomous trading agents. In *Proc. of the Second Int'l Conf. on Autonomous Agents and Multi-Agent Systems*, 2003.