Arena : Simulating E-Commerce Agent Strategies

by

Peter Ree

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 22, 2000

Author_____

Department of Electrical Engineering and Computer Science
May 22, 2000

Certified by_____

Professor Pattie Maes
Thesis Supervisor

Accepted by_____

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Arena : Simulating E-Commerce Agent Strategies
by
Peter Ree

Submitted to the
Department of Electrical Engineering and Computer Science

May 22, 2000

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Electrical Engineering
and Master of Engineering in Computer Science and Electrical Engineering

ABSTRACT

Arena is a software package intended to help e-commerce strategy researchers and designers test buying and selling strategies under different scenarios. It implements and adheres to a set of design criteria intended to ensure its simulation ability as well as prolonging its usable lifetime. The process of implementing Arena involved conducting background research into current strategies and the requirements they impose on a simulator as well as implementing a simple prototype simulator. Arena was used to evaluate aspects of Sardine, a system which models an airline selling tickets to potential buyers. Sardine showed Arena proficiency in implementing systems that provide meaningful results. However, Sardine also revealed some minor weaknesses in Arena's design that would benefit from additional research.

Thesis Supervisor: Professor Pattie Maes
Title: Director, MIT Media Lab Software Agents Group

Table of Contents

1 Introduction

2 Design Criteria
2.1 Simulation Ability Design Criteria
2.2 Lifetime Design Criteria

3 High Level Arena Design
3.1 Background Research
3.2 Satisfaction of Simulation Ability Criteria

4 Arena Implementation
4.1 Java
4.2 Development Process
4.3 Implementing a Simple Simulator
4.4 Simple Design

5 Implementation Details

6 Key Algorithms

7 Implementing a Scenario with Arena

8 Sardine Implementation and Arena Revision

9 Conclusion

10 References

1 Introduction

The evolution of the World Wide Web has lead to many remarkable changes in the way commerce is conducted in today's Digital Age. It's ability to provide a vast amount of information in an accessible form is perhaps its most celebrated feature and is often times also credited for helping e-commerce to grow. Web savvy consumers can now make better informed purchasing decisions by using the Web as a tool. However, the Web's wealth of information is only one feature among many that has allowed electronic commerce to flourish. New user interfaces, business models, and software have made significant and continue to make significant contributions to e-commerce. A common theme motivating many of these new technologies is increasing the ease of use for a consumer when making a Web based purchase. One only has to compare the Web experiences of users in the near past to see how today's users benefit from such technologies. Today's Web consumers now have one click ordering, electronic package tracking, and personalized accounts at their disposal, all of which help to simplify using the Web to make purchases. This evolution is far from over and we are beginning to see the next step in changing the look of e-commerce. The trend of improving ease of use is beginning to take a new form, as *automation* has become more prevalent in e-commerce.

One interesting approach to providing increased automation can be seen in *agent* based e-commerce systems [1]. We are presented with a good example of this in *Market Maker*, a system that uses agents to represent buyers and sellers in a market [2]. Users wishing to

buy or sell, create agents by instantiating them with their desired parameters (such as start price and price changing behavior). Once the agents are created, they are released into the market. These agents then begin to negotiate with each other until an agreeable price is found, which is then reported to both parties involved in the transaction. We can easily see that this system provides many benefits for a wide variety of users. A person wishing to purchase something on the Web may be too busy to do the price negotiating him/herself, or may not be familiar enough with the Web to make bids on an item. Another good example of a real world application of agents helping to solve this problem can be seen in ebay's simple bidding agent which automatically make a bid if some other person or agent has the current high bid [3]. The agent continues to place bids until it has either won the auction or the price of the item surpasses the maximum amount the person is willing to pay. Thus this simple agents allow a person to be represented in the auction when they might not have the time to sit at their computer or may not have the patience to continuously check to see if someone has outbid them. It is not difficult to see that electronic commerce agents can ease these and many other difficulties.

However, it is difficult to specify certain aspects of an agent's behavior and this is a problem my thesis addresses. Much of an agent's success is not only dependent on its own strategy but also on the strategies other agents employ. While the simple ebay agent provides a certain degree of usefulness, how will this change as automation is increased and new agents are developed? How will this simple agent fare against other, more complex, agents when competing for some item offered on ebay? In addition, how can selling agents best perform when negotiating with varying types of buyer agents using

different kinds of buying strategies? It is immediately apparent that agent strategies will have a large influence on how e-commerce is conducted in the future. In recognition of the importance of agent strategies, new research has begun in investigating the effect of different kinds of strategies on markets as well as the success of individual agent strategies. Much research is currently exploring dynamic pricing strategies where an agent can quickly change its behavior in response to a change in its environment. For instance, an agent may wish to alter its pricing behavior in response to its inventory or in response to a changing search cost [4] [5]. However, studying these strategies requires a tool to help understand their behavior. My thesis explores a simulation based approach to studying e-commerce agent strategies. This paper discusses the design as well as the implementation of *Arena*, a versatile and flexible strategy simulator that provides strategy designers and e-commerce researchers with a tool for observing the behavior of different agents.

A simulator is not the only solution to help our understanding of agent strategies. In fact a more exact and accurate solution would be to mathematically model different strategies and to derive observations through calculations. However, this is by far no simple task and presents many barriers to entry to those that would like to study agent strategies. The mathematical approach might not even produce results for a prolonged time due to the very complex nature of the problem. For instance, there exists both an infinite number of scenarios to arrange buyers and sellers and an infinite number of strategies each buyer and seller can use. While a great wealth of economic theory already exists that may provide some insight into these calculations, there is no guarantee to how useful these

theories will be when modeling this new Digital Economy. In contrast to the mathematical approach, a simulator presents a simpler and more intuitive means for conducting similar studies. Instead of laboring over abstract mathematical representations of buyers, sellers, and their strategies, literal representations of each can be simulated in a model that represents real world interactions. While the results from these simulations may be less accurate than the mathematical approach, the intuitive means will make the study of agent strategies available to more people. This may in fact lead to new theories that help simplify the mathematical approach

2 Design criteria

Any successful simulator should conform to certain design criteria to help ensure its success. However, it is important to note that the success of the simulator cannot be judged simply on its ability to accurately simulate many different scenarios of buyers, sellers, and their strategies. Success should also be defined by an ability to remain useful to many different users for a sustained period of time. That is, in the future, we should not have to duplicate the entire design and implementation process due to any of the simulator's shortcomings. From these observations we can establish that a successful simulator design should satisfy two sets of criteria: one to ensure its actual simulation ability and another to ensure its useful lifetime.

2.1 Simulation Ability Design Criteria

This set of design criteria is composed of 3 main objectives that a simulator must satisfy.

1. Modeling Flexibility
2. Behavior Support
3. Statistics Gathering

Modeling Flexibility

This is perhaps the most important of the 3 criteria because it details the scenarios a simulator can and cannot model and thus directly affects a large part of its success. However, it is impossible to account for every single possible scenario and so at the minimum, a simulator should be able to account for the most common and most useful scenarios. The simulator should then strive to attain coverage of as many possibilities beyond these requirements without sacrificing any ability to simulate the common cases.

Behavior Support

It is insufficient to simply design the simulator to account for a variety of scenarios. While satisfying this criteria is necessary, we also need to ensure its ability to handle different agent strategies. In order to do so, the simulator must provide a means for a multitude of strategies to function properly. This includes providing a simple means for a strategy to access its required decision variables. However, similar to the problem of an existence of an infinite number of scenarios, the simulator should first ensure that it can support the most common and useful strategies before attempting to support other strategies. Once this is done, it should then try and account for as many different types of strategies as it can.

Statistics Gathering

In order to analyze the simulator's results, it needs to output useful and correct statistics. The statistics are important to the user since they will help the user interpret the scenario being modeled. A perfect simulator that implements a high degree of flexibility and behavior support is useless if it outputs faulty and trivial statistics. The different types of statistics that a user may wish to gather also covers a large range of possibilities and so it is important to account for the many types of statistics as best the simulator can.

2.2 Lifetime Design Criteria

This set of criteria can be mostly satisfied through the observation of sound programming and software design principles. However, it is useful to explicitly state in this section which principles we should pay particular close attention to.

Usability

The sole purpose of the simulator is to provide many users with a tool to learn more about the behavior and performance of different strategies. Even if the simulator satisfies all of the above simulation criteria, a complicated mechanism for configuring a scenario and or testing a strategy will limit its success and may even render it useless. Thus the simulator must provide a reasonably simple means for allowing a user to both configure the simulator and build strategies they wish to test.

Modularity

Decomposing the design into several well thought out components will help the system to survive over time. As better implementations for a component or different user needs arise, modularity will help address these changes and allow them to extend the simulator's capabilities with minimal change to the rest of the system. For instance, if a faster algorithm is needed for a certain module, it can be implemented in a new module which can be then exchanged with the old. Also, if additional capabilities of a module are needed, these can be implemented in a new component. It is easy to see how modularity will help with the "plug-in" nature of testing different strategies with the same simulator. In this manner, we can use one simulator to test many different strategies.

Generality and Interface Use

Modularity will help when well defined and foreseeable components are needed. However, the software design should also account for and anticipate the changing of components. For these changing components, a robust set of abstract programming interfaces is required to help component designers interface their work with the simulator. Interfaces will allow designers to understand the simulator at a high enough level to enhance efficiency but will also provide them with a useful set of tools. From the perspective of the strategy designer we can easily see how interfaces will help make his/her work easier. A provided interface for the strategy and the simulator will help the designer quickly develop the strategy itself, instead of having to have full knowledge of the operation of the simulator.

3 High Level Arena Design

This section will discuss a high level overview of Arena's design and how it satisfies the criteria for a successful simulator outlined in the previous section. However, the evaluation of Arena's satisfaction of the criteria was done through two kinds of research: background research and the implementation of a simple system. This section discusses how conducting background research helped design components to satisfy the design criteria.

3.1 Background Research

Much of my thesis research was done through examining current strategy research and agent based e-commerce system. This research began with working with David Wang's *Market Maker* system. Through this work, I was exposed to a real agent based e-commerce system and was able to gather a great deal of knowledge of the interactions between buying and selling agents. Another valuable experience was in an earlier project of mine at the Media Lab which explored an agent based approach for a business to business hub. This also helped solidify the requirements for the infrastructure of any system in which buyers will interact with sellers. Many of these requirements in fact appear in Arena's architecture, which will be discussed in a later section. Another helpful resource my research benefited from was as Jeff Kephart's IBM Economics Agents Group. Kephart's work was extremely useful in seeing the types of strategies people were interested in testing and also in how they were tested. This helped to

illustrate the capabilities a simulator should have in order to suit these strategies [6]. The strategies themselves helped to give me a better understanding of the degree of complexity strategies have. This exposure to complex strategies helped to extend the strategies I have been exposed to through my background in economics. Much of this background includes a knowledge of game theory which helped me to better understand how many scenarios can in fact be modeled as a game.

From this great wealth of information, I constructed a design for Arena which I thought best satisfied the outlined simulation ability design criteria.

3.2 Satisfaction of the Simulation Ability Design Criteria

Modeling Flexibility

From my research, I discovered that this criteria would be best satisfied by constructing an environment that could be configured to account for as many scenarios as possible. I tried to find a common means for unifying the different types of commerce type transactions and strategy based negotiation models came across. I found a means for unifying all of these models through the use of several key entities. Each entity in this configuration was derived from this research and is included in Arena's simulating environment. They are:

- A variable number of buyers, sellers, and types of goods

The Arena environment can be configured such that a simulation between any number of buyers and goods negotiating over different types of goods can be used to test strategies.

- Different types of buyer and seller strategies.

Arena can support different types of strategies for buyers and sellers interacting with each other.

- Different types of goods being sold in the environment.

Sellers can offer multiple types of goods and buyers can be interested in more than one kind of good.

- Different types of negotiations

Negotiations can be single buyer to single seller or between multiple buyers and sellers. Negotiations can also involve different types of goods.

- Different types of simulation behavior

The behavior of the simulation with respect to it state can change similar to how buyers and seller behavior can change as their strategy dictates. For instance, Arena can vary the number of buyers that a seller has the opportunity to sell to if time in the simulation has surpassed a certain point. This may be useful in such scenarios to reflect changing market conditions, such as an increasing amount of information available to buyers. Such a model can be found when examining the effect of the increasing number of Web savvy people and online sales.

- A notion of time

Arena's environment has the ability to keep track of time. That is, time as seen by the buyers and sellers involved in the simulation. My research showed that maintaining time in the simulation state was a necessity since many strategies as well as simulation behaviors depend on time.

- Simple statistics

Arena's architecture allows for all events and transactions to be logged. This provides good coverage of the statistics that a user may be interested in.

These design components all help to satisfy the simulation ability set of criteria. The satisfaction of the lifetime criteria will be discussed in the next section, along with a more in depth look at Arena's implementation of the components listed above.

4 Arena Implementation

This section will provide a more in depth look at Arena's implementation while also justifying the design decisions made. The approach in this section is more from a software level and thus will provide insights into the actual code implementation as well as how the lifetime criteria is satisfied. From the following sections, it will become apparent how the lifetime criteria is satisfied through the use of an intuitive structure for the simulator, interfaces that facilitate the testing of different strategies, and a modular design to help minimize the effort required to modify and maintain Arena.

4.1 Java

To further enhance Arena's usable lifetime, Java was the language chosen for implementing Arena. While a faster, native language could have been used, such as C or C++ which have native compilers, I feel that Java will help others to use Arena. Not only at the Software Agents Group, but also at the Media Lab in general, most undergraduates as well as a good deal of graduates seem to be programming in Java. It also seems that in general, an increasing number of people are using Java to write their research software at the Media Lab. Writing Arena in Java would help others who are new to it and wish to use it understand it quickly. Using programming language common to many people will also promote its maintenance in the future.

4.2 Development Process

Arena's development process was composed of 3 main stages:

1. Implementing an initial simple simulator
2. Designing and Implementing Arena
3. Revising Arena after implementing Joan Morris' research project

At each stage, a significant amount of useful findings were gathered and helped evolve Arena into its final form.

4.3 Implementing a Simple Simulator

Stage 1 involved implementing a simulator to model a simple scenario of buyers and sellers. In this scenario, buyers were interested in the type of good all the sellers were offering. A seller was selected from the pool of sellers. This seller would then locate a certain number of buyers (a subset of all buyers) and query each to see if they would like to purchase from the seller. This decision was based on the seller's offering price and the buyer's purchasing price. Successful transactions were noted, and the simulation would then continue by selecting the next seller.

I learned a great deal from this simple model regarding the issues I would need to address in implementing a more advanced and flexible system. This was done by studying the components of the model that made the model itself simple and anticipating how allowing these to be more complex could be implemented. These components were:

Search Width
This value is the number of buyers the seller "finds". Thus a seller that can reach more potential customers will have a larger search width than one that cannot. In my simple system, this was implemented as a Gaussian distributed random number, center at some parameter passed into the system. A more complex

system should be able to better specify how each seller finds a potential customer. For instance, a seller may wish to only search for buyers that satisfy a certain criteria, such as finding a target group based on age or gender.

Active Searchers
A closer examination of a seller's search width revealed that not all seller's search for buyers. In fact, often times it is the buyer searching for a seller (such as in a mall or a farmer's market). There also exists many real world examples of both parties searching for each other. For instance, many online retailers will send out promotions to potential customers via email or some other means while many potential customers may also be currently searching for these retailers.

Goods
The simple scenario involved sellers selling a universal "good" which all buyers were interested in. A more complicated and realistic scenario would conceivably involve multiple types of goods or similar but slightly differentiated goods. Likewise, a more flexible simulator should be able to model scenarios with sellers offering bundled packages of different types of goods.

Negotiation Terms
In this simple model, price is the only factor that determines if a buyer will purchase from a seller. That is, only the seller's offering price and the buyer's reserve price figure into any kind of negotiating between both parties. A more flexible simulator should be able to model more realistic scenarios where negotiations can involve more than just price. Other negotiation terms should be able to be handled if the scenario requires them. This would better reflect real world situations where buyer often concerned with many factors aside from price. Sellers in the simulation will also benefit from this ability since real world sellers often differentiate themselves from others by offering different prices as well as different negotiable terms to a product.

4.4 Simple Design

The issues that the simple scenario raised, together with the components outlined in the satisfaction of simulation criteria section all made up several pieces of a flexible simulator system. The next task in the development of Arena was to find a means for connecting these pieces to form a flexible system that satisfies both the simulation and lifetime criteria.

The simple model discussed earlier had a very important structure for linking the buyer and seller components in that it was simple and intuitive. I wanted to retain this simplicity in Arena's design for two reasons. The first being that a simple and intuitive design would be easier to implement since I could take cues from the real world to help conceive as well as design the interactions between components as well as their behavior. The second reason was that an intuitive model would be simpler for future maintenance and use by other Arena users. Components of Arena represent real world objects that are familiar to many people and will help future users understand how the system operates without requiring an intimate knowledge of the software implementation. For instance, I could have implemented Arena as a system "X" made up of one single component that could not be decomposed into separate, independent components. However, maintaining "X" would be difficult since it lacked a comprehendible structure. Instead, if I specified that Arena was composed of "buyers" "sellers" and "negotiations", the model's operation is immediately more understandable than "X"'s. This intuitive design greatly aides in satisfying the lifetime criteria of a successful strategy simulator.

In fact, this is exactly how I achieved Arena's intuitive and simple structure. It's behavior operates around 3 key components that reflect real world transactions. These three components are:

1. Buyers
2. Sellers
3. Negotiations

However, the intuitive structure of Arena pertains more to the state of the simulator rather than to the entire simulator itself. That is, each instance of these three key components combine to form a state which by itself does not simulate a scenario. Another key component to Arena is used to manipulate the state so actually run the simulation. This component contains the main algorithms for the execution of the simulation and dictates how the simulation state changes. However, in order to maintain a safe abstraction barrier between this component and the simulation state, a third component is used as an interface between the two. Thus Arena's high level structure can be viewed as being made up of three layers:

1. The Engine : simulation algorithms
2. The State Changer : handles all requests to manipulate the simulation state
3. The Simulation State : maintains the intuitive model of buyers, sellers, and negotiations.

This also details the hierarchy of Arena's structure. The Engine commands the State Changer which in turn manipulates the Simulation State. This three tier structure helps satisfy the lifetime criteria in that any future changes to Arena in any of these three components will have minimal impact on the other. If in the future a different simulation algorithm is needed, then the Engine component can be rewritten or extended to incorporate new features. Since the Engine is de-coupled from the State Changer and the Simulation State, they will not need to be changed.

5 Implementation Details

This section will provide an in depth review of some of the key classes in Arena's structure.

Simulation State

This class represents the state of the simulation and encapsulates all of the smaller components of the simulation state such as the buyers, sellers, and negotiations. The advantage of collecting the components in one class is that it allows Arena's structure to have a definite *state* it can refer to. An alternative would be to let each of the smaller components manage themselves. However, this loosely joined structure for the state would make all state manipulations involving different types of components extremely inefficient to execute. Multiple requests would be dispatched to each of the components which would need to be located in the system. The Simulation State class provides a simpler and more efficient means for carrying out such requests. Instead, one request for a state manipulation can be passed to the Simulation State which has specific knowledge and control of each of the smaller components. This request can be then efficiently handled as needed.

The Simulation State will also aide in any future additions to the system's state. Any new component that is added will be added to and managed by the Simulation State. Most likely, any new component will also introduce new interactions with existing

components. The Simulation State centralizes all such interactions and so adding a new one to the existing set of interactions will not complicate any of the design structure.

Transaction Party Interface

This interface represents all methods that any party involved in a transaction should have. It is used to reflect that an object implementing the interface can be part of a transaction. This is extremely useful in that two transaction parties negotiating a transaction will know exactly how to communicate with each other. For instance, a party's identification will often be needed for verifying identity and one transaction party can call the getID() method of the other transaction party. Other more complex methods are also included, such as for querying the party if it would like to negotiate with a certain other transaction party. Defining and interface instead of creating a separate class allows several classes to implement each interface method as they chose. This is important since some parties may wish to implement these methods differently. Currently, the only such parties are buyers and sellers. However, in the future, it is conceivable that new types of transaction parties will be introduced into Arena's environment.

An alternative choice would be to not use an interface at all and instead let each object have its own proprietary interface for communicating during transactions. However, the disadvantage of this scheme is that any new party that is added and wishes to be part of a transaction will have to notify all other parties of its proprietary interface. Not only will these parties need to record this special interface, they will also have write new tests for when to use this interface. Each transaction party will have to have a library of interfaces

and an appropriate test that is used when dealing with another transaction party. This test will determines which interface in the library to use. Using an interface allows all transaction parties to treat different classes in a general manner. No testing is required since all can be addressed by the Transaction Party interface.

Active Searcher Interface

This interface describes parties that actively search for their counterparts so that a transaction can take place. This allows both buyers and sellers to exhibit this ability and does away the need to implement two types of buyers and sellers: a buyer that actively searches for a seller and one that does not and also a seller that actively searches and one that does not. This allows buyers and sellers some degree of specificity that would be degraded if these additional variants of each would be needed. Similar to the advantages of the Transaction Party interface, the Active Searcher Interface also provides all components that communicate with an active searcher a general interface to use and removes any need to keep a library of interfaces. Thus many different kinds of active searchers can be efficiently implemented with little need to change or update other related components.

Buyer and Seller Interfaces

These interfaces describe buyers and sellers in the Arena simulation environment. Using an interface rather than a specific and concrete class for both components is vital for the flexibility of Arena. It gives a common means for communicating with different types of buyers and sellers. For instance, two different buyers implementing vastly different

characteristics can be treated in the same manner. As with the justification for the above interfaces, this allows buyer and seller designers to efficiently develop these components for a simulation scenario. Since it is the goal of Arena to be able to model many different scenarios, the use of buyer and seller interfaces reflect anticipation of many different implementations for both components. A hard coded alternative would be to instead implement both as concrete classes. However, this mistake would definitely cause problems in the future. For instance, hard coding a buyer binds all future users to that implementation and greatly restrict the capabilities of the simulator. An interface based approach in fact promotes the development of capabilities since designers can assume that their buyers and sellers will be compatible with Arena if they correctly implement the Buyer and Seller interfaces.

Strategy Interface

Similar to the use of Buyer and Seller interfaces, the Strategy interface is used rather than a concrete class since many different strategies will be used. As discussed earlier, this provides all the advantages of modularity and a component based design in that a strategy components can be exchanged with little effort needed for integration. This feature is very powerful when you consider that Arena's intended use is to test strategies. A user can simple swap different strategies in and out of the system for testing, while leaving the remaining components untouched.

Negotiation

This class encapsulates a negotiation between at least one buyer and at least one seller in the simulation state. It serves as a repository for all information that is public during a negotiation and thus buyers and sellers can query it to obtain knowledge about the negotiation taking place. In addition, the Negotiation can be used to send messages to all parties involved in the negotiation. This is efficiently done via the Observer object oriented design model [7]. A nice feature of this model is that it allows for different types of messages to be easily added. A new message only needs to be added to the Negotiation class. When this message is sent out, all parties will receive it. Any party interested in the message will then behave accordingly after receiving the message, all uninterested parties require no modification since they do not wish to respond to it.

An alternative solution would have been to allow buyers and sellers to simple handle negotiations themselves. While this may seem a valid solution since many scenarios can be modeled in this manner, it limits the possible types of negotiations that can be modeled and puts a tremendous amount of responsibility in the buyers and sellers. For instance, if we wanted to model a specific buyer in an environment where the buyer negotiates with a seller, this model would work fine. However, if we then wanted to test this buyer in a collectively purchasing environment where many buyers collaborate to make purchases (such as in the now popular web sites www.mercata.com [8] and www.mobshop.com [9]), we would have to make extensive modifications to the buyer to handle communication with the seller and its fellow buyers. An even more complicated scenario would be if we then added multiple sellers to the negotiation. Providing a

negotiation class allows us to remove this burden from both the buyers and the sellers. It allows Arena to provide more flexibility in the types of negotiations possible as well as for maintaining the specificity of buyers and sellers (i.e. they encapsulate buying and selling, not the many types of possible negotiation processes).

Engine

As discussed earlier, the Engine serves as the driving force of the simulation. It houses the main simulation algorithms for simulation and dispatches commands to the State Changer to manipulate the Simulation State. This helps to somewhat localize the simulation algorithms. While not all of the simulations algorithms are found here, many are also contained in the interactions of the many components of the Simulation State, the key algorithms for beginning negotiations are. This is perhaps one of the most important events in a simulation. Many strategies will depend on the other party or parties involved in the negotiation and these algorithms will dictate who becomes joined in a negotiation. It is important that these algorithms be isolated so that modifications to them will require minimal change (if any) to the other components of Arena.

State Changer

This class acts as a proxy for all components that wish to modify the Simulation State in any way. While this currently provides only a small advantage buy adding a clear abstraction barrier between the Simulation State and the other components of Arena, its inclusion in Arena's design was with the future in mind. A situation may arise where a simulation designer wishes to model a scenario that is beyond Arena's capabilities,

however, this scenario can be attained if the Simulation State is augmented. The State Changer would then act as a proxy for both components and so this addition will be hidden from all components using the State Changer.

6 Key Algorithms

The previous section outlined the key components of Arena and discussed their responsibilities as well as the behavior. This section will show how they interact with each other in Arena's structure. The clearest means to illustrate the many interactions is to present Arena's simulation algorithm.

At the highest level, we have the Engine's simulation **while** control loop. This loop continues until some specified stop criteria is met. During each iteration of the loop, the general idea is to allow each Active Searcher to try and find their counterpart. That is, each actively searching Seller is given a chance to find Buyers interested in purchasing its product and each actively searching Buyer is allowed the same opportunity to find Sellers. If a match is found, then the State Changer requests that a Negotiation is started between the two matched parties. After all Active Searchers have been given a chance to search, the simulation time is then incremented.

Engine **while** control loop pseudo code:

```
1.  while the stop criteria has not been met do
2.      aso = the active searcher order retrieved from the State Changer
3.      while there are more active searchers in aso do
4.          active = the next active searcher in aso
5.          potentials = the parties active is searcher for, which is retrieved from the
State Changer
6.          while potentials has more Transaction Parties do
7.              tp = the next Transaction Party in potentials
8.              if tp's interests matches active's interests do
9.                  ask the state changer to make a transaction between tp and active
10.             end if
```

11. **end while**
12. **end while**
13. State Changer increments time.
14. **end while**

Here we notice that the simulation designer is given a good deal of control over how the simulation is executed. Different criteria for ending the simulation can be given as well as the order in which the active searchers are allowed to search. This is useful in that the designer may be looking for a specific event to happen and may wish to end the simulation upon seeing this event. The stop criteria gives the designer this freedom. Also, the active searcher order allows the designer to implement varying capabilities of Active Searchers. For instance, perhaps an Active Searcher more adept at searcher is given two opportunities in the search order to search or perhaps the search order changes in response to how a currently searching Active Searcher performs.

Line 13 of the **while** control loop is of special importance. Incrementing time signifies that a round of simulation time has ended. It is important to note that although the State Changer requests the Simulation State to increment time, the round is note quite over yet. In the pseudo code for the Simulation State's method to increment time, we see that quite a bit occurs until the actual time in the Simulation State is changed. Two phases occur, the first being prior to the increment and the second being after time has been incremented. In the first phase, the Simulation State asks each of the currently active Negotiations to continue negotiating. The algorithmic detail of the negotiating is heavily dependent on the strategy of all parties involved in the negotiation. The overall goal of asking a Negotiation to continue negotiating is to get all parties to strive closer to an agreement, or to discover that the negotiation is dead. In either the case of a successful of

failed negotiation the simulation state is notified so that it can make the appropriate changes. In the second phase of the time increment, the Simulation State conducts a cleanup of itself. Buyers and Sellers with expired lifetimes are removed from the current pool of buyers and sellers. Also, they are removed from any active Negotiation they are involved. The Negotiation is removed as well if it does not have a sufficient number of parties (i.e. at least one buyer and one seller) to continue.

Simulation State increment time pseudo code:

```
1.  notify each Negotiation to continue negotiating
2.  increment time
3.  remove all Buyers from the current pool of Buyers if their lifetime has expired
4.  if this Buyer was involved in a Negotiation do
5.      remove the Buyer from the Negotiation
6.      if this Negotiation does not have enough parties to continue do
7.          remove the Negotiation from the currently active Negotiations
8.      end if
9.  end if
10. remove all Sellers from the current pool of Sellers if their lifetime has expired
11. if this Seller was involved in a Negotiation do
12.     remove the Seller from the Negotiation
13.     if this Negotiation does not have enough parties to continue do
14.         remove the Negotiation from the currently active Negotiations
15.     end if
16. end if
```

7 Implementing a Simulation with Arena

This section will outline the steps needed to implement a simulation using the Arena Java classes. However, this section only provides the general steps that should be taken. The reader is encouraged to examine the class definition files for a more specific description of the general topics discussed here. Before proceeding, it is important to note the distinction between the designer's model and the Arena simulation he/she wishes to build. A "model" refers to the outline of how Arena should be configured to *represent*. Depending on the capabilities of both Arena and the designer's proficiency with Arena, a simulation will have varying accuracy of representing the designer's model. Building a simulation consists of 4 main steps that should be done in this order:

1. Goods sold/sought in the simulation
2. Representation of time
3. Active Searchers
4. Buyer and Seller implementation

Step1 : *Goods sold/sought in the simulation*

In this simple step, the simulation designer finalizes the goods being sold by sellers and sought after by buyers. It is necessary that *all* seller offerings and buyer interests are well understood before proceeding on to Step 2. A designer must know what each buyer is interested in buying in the simulation and what each seller is offering in the simulation. This also includes implementing a Matcher to determine whether or not two Transaction Parties are compatible (one is selling something the other is interested in).

Step 2: *Representation of time*

A key to designing a simulation is to know the role of time. Given the Engine's **while** control loop, each increment of time will cause certain deterministic actions to occur. Thus the designer must know how these actions should effect the simulation state. Arena helps to simplify this by not associating any unit with the time kept in the Simulation State. However, it is important that the designer know how one unit of time in the Simulation State corresponds to one unit of time in the designer's model.

Step 3: *Active Searchers*

In any simulation, at least one Buyer or one Seller must implement the Active Searcher interface so that at least one party will seek out its counterpart. An Active Searcher can be a Buyer or Seller and should reflect how the party behaves in the model the designer is simulating with Arena. A buyer or seller in the model which actively looks for another party should implement this interface in the simulation.

Step 4 : *Buyer and Seller implementation*

This step requires that the Buyer and Seller interfaces are implemented by some concrete classes to represent the buyers and sellers of the designer's model. This also includes implementing their strategies as well as determining their lifetimes.

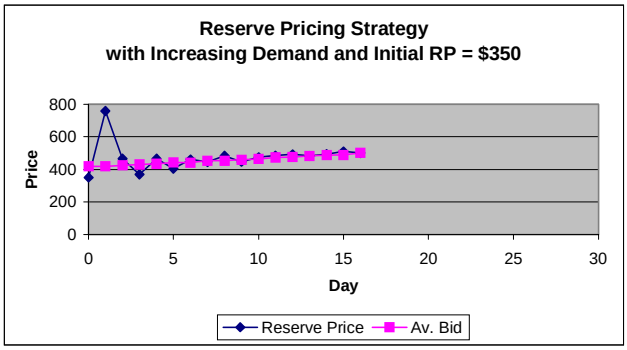
Once all four steps have been completed (again, these are very general steps that need to be accomplished), they can be incorporated with the Engine and the simulation is ready

to be run. However, additional classes are recommended to facilitate easy parameter passing to run different experiments with the simulation.

8 Sardine Implementation and Arena Revision

After completing the implementation of Arena, my next task was to see how well it did in practice. Its first trial would be in implementing the seller strategies in *Sardine*, Joan Morris' airline strategy scenario [10]. In this scenario, a single airline seller wishes to sell a certain quantity of airline tickets to a population of buyers by a certain deadline. Every day, a certain number of buyers offer the airline bids for tickets. If a given bid is above the airline's reserve price for that day, then the ticket is sold to that buyer, if not the buyer is unsuccessful in obtaining a ticket. Morris wished to examine the effect of different market conditions on different classes of selling strategies. So in configuring Arena, I built a driver class for Morris' simulation which created the pool of buyers and the airline seller based on a configuration file. These buyers extended the Buyer interface and implemented only the necessary methods. Likewise, the airline was implemented as the Seller.

Here we see an example of Sardine's results for simulating a simple pricing strategy based on the total quantity sold to date. As demand increases, the airline adjusts the reserve price to the bid level and then follows its increase [11].



However, in implementing Sardine, I noticed a slight problem with Arena's architecture. Specifically, statistics generation was not quite as simple as I had previously thought. Initially, I thought that the statistics could be stored with each significant component. This assumption turned out to be quite naïve in that much effort was required to collect each components repository of statistics. Although Sardine operated successfully with this cumbersome statistics mechanism, I decided to revise Arena and add a simpler, more efficient means for statistics gathering. Instead of scouring each component for their accumulated statistics, a single repository is used to collect statistics as they come. This architecture vastly improves Arena's statistics generation ability.

9 Conclusion

Morris' work with Sardine produced some interesting findings and presents us with some insight into seemingly simple airline selling strategies. Morris found that using a strategy to change the reserve price of a ticket based on the quantity sold to date has an ability to track the demand level quite well. This simple strategy which indirectly tracks demand in fact outperforms a slightly more complex strategy that adjusts the quantity of seats released based on the current demand.[i] This adjustment was an attempt to optimize the number of tickets offered each period and thus increase the airline's profits. Sardine has shown us that our assumption that we can engineer better strategies is not always correct. This realization illustrates the power of simulation. What we also see is that Arena succeeds in providing a powerful mechanism for strategy designers to test their ideas. However, Sardine also reveals some areas in which Arena can be improved.

Specifically, Sardine suffered from a poor user interface. Parameters were passed into the simulation via a configuration file. While this scheme worked fine, and would suit most other simulations as well, this is far behind the user interface standards of most applications. Ideally, Sardine would be equipped with a simpler means to enter in data such as a succession of windows and help menus. Since it is difficult if not impossible to include a graphical user interface package with Arena due to the infinite possibilities of scenarios Arena can model, all Arena simulations will suffer from this problem. All will require that some user interface is developed on its own. This may in fact detract from the satisfaction of the lifetime criteria established earlier since implementing a user

interface will increase the amount of effort for implementing a system. However, I believe that this problem can be tolerated since user interface design is a well understood problem and many software libraries currently aide this development and thus Arena's user will have ample amounts of assistance.

However, one problem that Arena would definitely gain from its improvement is the necessity of recompiling the Java source file for every new simulation. This problem stems from the decision to use Java, which does not support dynamically linked libraries as Arena's language for implementation. One possible solution would be to devise a scenario-strategy language specifically for Arena. Conceivably, this language would able to describe all the scenarios as well as all the agent strategies Arena can support. If this was achieved, then this language could be used to dynamically configure Arena. A scenario-strategy source file could be parsed and then used to configure Arena's class file which would then execute the defined simulation. In this manner, Arena would not have to be repeatedly compiled. However, currently recompiling Arena is not a significant problem. While this may change in the future, devising the scenario-strategy language is a definite major endeavor which would require much research.

However, despite these minor shortcomings, we see in Sardine that Arena provides a powerful tool for e-commerce agent strategy designers and researchers. In comparing seemingly simple strategies against engineered strategies, Morris' work shows us that our understanding of strategies may not quite be as intuitive as we think. These findings are from one of many possible applications of Arena and only hint at Arena's usefulness in

future projects. It is when we become close to realizing Arena's full potential that the e-commerce community will posses a better understanding of a wide variety of buyer and seller strategies. With this knowledge in hand, the importance of agents will continue to flourish and hopefully we will have Arena to thank for helping to continue the evolution of agent based e-commerce.

10 References

1. Maes, P., Guttman, R., and Moukas, A., "The Role of Agents as Mediators in Electronic Commerce." Special Issue of Knowledge Engineering Review on Practical Applications of Agents, Edited by Barry Crabtree, summer 1998
2. Wang, David. http://ecommerce.media.mit.edu/maker/maker.htm. 1996
3. http://www.ebay.com
4. J. Morris. P. Ree, and P. Maes. Dynamic Seller Strategies in an Auction Marketplace. Submitted to the ACN Ecommerce Conference, 2000
5. J. Kephart, J. Hanson, and A. Greenwald. Dynamic Pricing by Software Agents. In *Computer Networks*, March 2000.
6. J. Kephart, J. Hanson, and A. Greenwald. Dynamic Pricing by Software Agents. In *Computer Networks*, March 2000.
7. Gamma, Eric, et al. Design Patterns : Elements of Reusable Object-Oriented Software. Reading, Massachusetts : Addison-Wesley. 1995.
8. www.mercata.com
9. www.mobshop.com
10. J. Morris. P. Ree, and P. Maes. Dynamic Seller Strategies in an Auction Marketplace. Submitted to the ACN Ecommerce Conference, 2000
11. J. Morris. P. Ree, and P. Maes. Dynamic Seller Strategies in an Auction Marketplace. Submitted to the ACN Ecommerce Conference, 2000

[i] J. Morris. P. Ree, and Dynamic Seller Strategies in an Auction Marketplace. May 2000.