

Supplier Agents within the MAGNET system

Stefan Botman

Mark Hoogendoorn

{snbotman, mhoogen}@cs.vu.nl

Department of Computer Science

Vrije Universiteit Amsterdam

Department of Computer Science and Engineering

University of Minnesota

16th December 2002

Contents

1	Introduction	1
1.1	A Motivating example	1
1.2	The MAGNET architecture	2
1.3	Current status of the MAGNET system	3
2	Related Work	5
3	Early design	7
4	Investigation of the Avalon tool	11
4.1	A brief introduction to Avalon	11
4.2	How Avalon works	11
4.3	Avalon and MAGNET	13
4.4	Conclusion	13
5	Designing the Supplier agent	15
5.1	Methodology	15
5.2	UML Use Cases	16
5.3	Design using DESIRE	31
5.3.1	Specifying the components	31
5.3.2	Defining the interfaces of the components	33
5.3.3	Information Exchange	40
5.3.4	Knowledge composition, information types	42
5.4	Design using UML	57
5.4.1	Class Diagram	57
5.4.2	Sequence Diagram	59
6	Implementation	65
6.1	Implementing the components	65
6.1.1	Implementing the information types	65
6.1.2	Working with Threads	67
6.1.3	Implementation of the AgentState component	68
6.1.4	Implementation of the BidManager component	70
6.1.5	ResourceManager, SalesAnalyst and PriceManager	71
6.1.6	CommunicationManager	71
6.1.7	SupplierAgent	72

7	Testing	75
7.1	BidManager	75
7.1.1	Queue	75
7.2	AgentState	76
7.3	The complete system	79
7.4	Conclusion	80
8	The SalesAnalyst	81
8.1	Analyzing the Sales Analyst	81
8.1.1	Proposals for learning schema's	81
8.1.2	Salesgoals	82
8.1.3	Some definitions	83
8.1.4	Necessary data	84
8.1.5	Communication with other components	84
8.1.6	Problems with the aforementioned proposals	85
8.1.7	Proposed solutions for the problems	85
8.2	Selecting the algorithm	85
8.2.1	The C4.5 algorithm	85
8.2.2	Reasons for choosing C4.5	90
8.3	Designing the SalesAnalyst	91
8.3.1	UML Activity diagram	91
8.3.2	Describing the tasks	93
8.3.3	Identifying the classes and the interfaces	96
8.4	Implementation	96
8.4.1	Feeding information to the C4.5 Algorithm	98
8.5	Running experiments	100
8.5.1	Setup	100
8.5.2	Results	101
9	Conclusions and Future Work	107
10	Acknowledgements	109
A	Paper for AAMAS '02 conference	115

Abstract

We are interested in supporting multi-agent contracting in which customer agents solicit the resources and capabilities of other self interested supplier agents in order to accomplish their goals. Goals may involve the execution of multi-step tasks in which different tasks are contracted out to different suppliers.

In this paper we focus on the design of supplier agents. The agents are designed to operate in the context of the MAGNET (Multi AGent NEgotiation Testbed) system, but the design could easily be adapted to other situations in which agents interact through a market infrastructure.

MAGNET supplier agents can register their capabilities with the market, be notified of open and relevant requests for quotations, and submit bids that specify which tasks they are able to undertake, when they are available to perform those tasks, and at what price. Supplier agents attempt to maximize the value of their resources.

This paper describes the detailed design of supplier agents and presents experimental results.

1 Introduction

Over the past few years a lot of research has been done on designing market architectures for electronic commerce. Almost all of these systems are able to handle simple agent-to-agent trading, which only address cost. Real life is however quite more complex than that; there should be a possibility to model other meaningful features to close a contract, these include time constraints, enforce deadlines, to interact with a highly distributed web of suppliers with different capabilities and resources, to interact over long periods of time through the completion of the contracted work, and to deal with failures in contract execution (see [Collins *et al.*, 2000]). In order to model these features the MAGNET (Multi-Agent Negotiation Testbed) system has been designed at the University of Minnesota (see [Collins *et al.*, 1998]).

1.1 A Motivating example

In order to make the working of the MAGNET system more clear, imagine the example of building a house. In figure 1 the tasks needed to build the house are represented in a task network (the links indicate precedence constraints).

Now the first decision that has to be taken is how to sequence the tasks in the Request for Quotes (RFQ), which is the request for bids on those tasks to be sent to *supplier agents*. It must also be decided how much time to allocate for each task. For instance, we could reduce the number of parallel tasks, allocate more time to tasks with higher variability in duration or to tasks for which there is a shortage of laborers, or to allow more slack time. The assumption has been made that suppliers are more likely to bid on, and submit lower-cost bids, when the flexibility in the scheduling resources is greater (see [Babanov *et al.*, 2002a]. This may result in overlapping time windows. An example of a RFQ is given in figure 1, as can be seen there is no restriction that the precedence constraints are satisfied within the RFQ, as long as they are satisfied with the accepted bids.

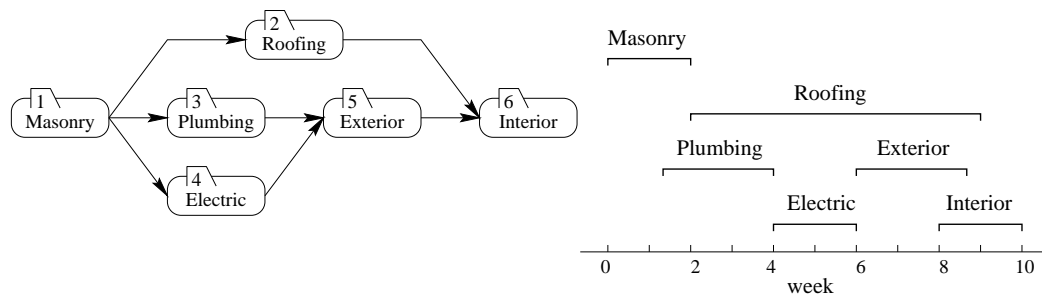


Figure 1: A task network example and a corresponding RFQ.

1.2 The MAGNET architecture

Let us give a brief introduction to the MAGNET System. It has been under development since early 1998 at the University of Minnesota. Two roles have been distinguished, namely the *customer* and the *supplier*. The *customer* is the agent who needs resources which he doesn't have direct access to. It's goal is to provide himself with the needed resources. The *supplier* is the agent who can provide the *customer* with the necessary resources. A design has been made to model the above stated requirements which is shown in figure 2 below. We will not go into further detail, for a complete explanation of the architecture, see [Collins and Gini, 2001b].

Now the main interaction between the agents is discussed.

- A customer agent issues a *Request for Quotes* (RFQ) which specifies tasks, their precedence relations, and a time line for the bidding process.

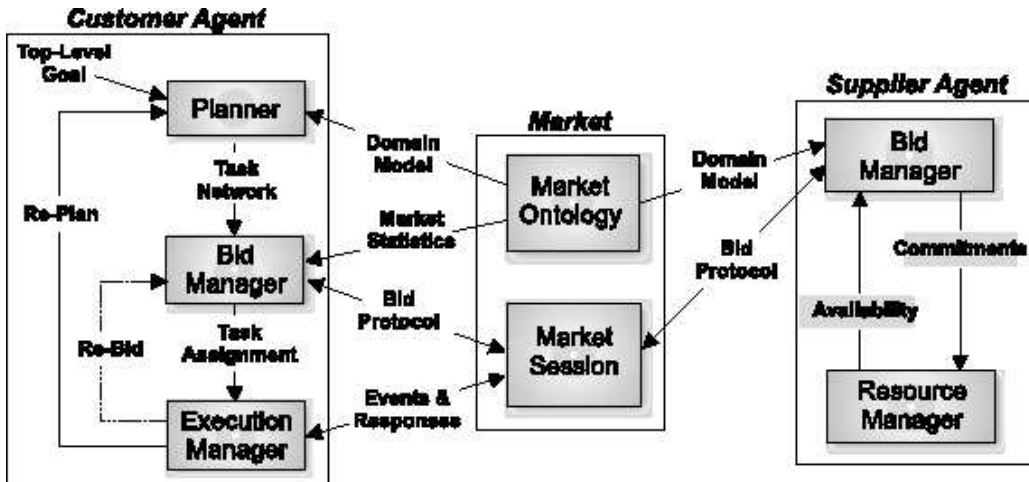


Figure 2: The MAGNET architecture

For each task, a time window is specified giving the earliest time the task can start and the latest time the task can end.

- Supplier agents submit bids. A bid includes a set of tasks, a price, a portion of the price to be paid as a non-refundable deposit, and estimated duration and time window data that reflect supplier resource availability and constrain the customer's scheduling process.
- The customer agent decides which bids to accept. Each task needs to be mapped to exactly one bid (i.e. no free disposal [Nisan, 1999]), and the constraints of all awarded bids must be satisfied in the final work schedule. In MAGNET the customer can choose from a collection of winner-determination algorithms (A*, IDA*, simulated annealing, and integer programming).
- The customer agent awards bids and specifies the work schedule.

1.3 Current status of the MAGNET system

Over the past few years, research on the MAGNET system has mainly focussed on the *customer* side, see [Collins and Gini, 2001a], [Collins *et al.*, 2002] and [Babanov *et al.*, 2002a]. However the *supplier agent* architecture hasn't been as thoroughly developed. In this paper the main focus will be

on creating a more sophisticated design of the *supplier agent*. In figure 2 there is already a simple design for the *supplier agent*, however this is a very shallow design and it will have to be extended in order to be able to model more advanced *supplier agents*.

2 Related Work

Markets play an essential role in the economy, and market-based architectures are a popular choice for multiple agents (see, for instance, [Chavez and Maes, 1996], [Sycara and Pannu, 1998], [Wellman and Wurman, 1998], [Tsvetovaty *et al.*, 1997], [Karacapilidis and Moraïtis, 2001], [Choi and Liu, 2001]). Most market architectures limit the interactions of agents to manual negotiations, direct agent-to-agent negotiation [Sandholm, 1996], [Faratin *et al.*, 1997], or various types of auctions [Wurman *et al.*, 1998].

Existing architectures for multi-agent virtual markets typically rely on the agents themselves to manage the details of the interaction between them, rather than providing explicit facilities and infrastructure for managing multiple negotiation protocols. As discussed above, agents interact with each other through a Market. Our Market infrastructures provides a common vocabulary, collects statistical information that helps agents estimate costs, schedules, and risks, and acts as a trusted intermediary during the negotiation process.

For the design of the supplier agent we've used component based design, already a lot of research has been done in design using components. For a thorough discussion of this type of design see for example [Szyperski, 1998]. Work on agent-based design can be found in for example [Kinny and Georgeff, 1997], [Brazier *et al.*, 1998] and [Padgham and Winikoff, 2002]. The approach taken in [Kinny and Georgeff, 1997] is similar to our approach when looking at the levels of abstraction used, however the methods used differ. In [Brazier *et al.*, 1998] the phases in the DESIRE design method are discussed, in our document the same method is used, however only for the components and the information flowing between them. The *generic agent model*, introduced in the same paper, isn't used either because in our opinion this would make the design of our agent more complex then it should be. [Padgham and Winikoff, 2002] introduces a method for design of intelligent agents which includes a lot of tools that can be used to help with the design and implementation of the agents. A similarity between the design they've introduced and our design is the use of interaction diagrams, however the other modeling methods are different from our approach. In [Shehory, 1998] a comparison between several different architectural frameworks including DESIRE is given, emphasizing the weak and strong points of each of these frameworks. Nowadays an extended version of UML has been proposed in [Bauer *et al.*, 2001] called Agent UML, since UML is commonly used this is an interesting development and

certainly related to our work. Since we use UML and it would have made life easier to have diagrams more suitable for agent design. However, because Agent UML is not standardized at the moment it has not been used. When implementing and designing agent-based software there are several pitfalls one can fall into, described in [Wooldridge and Jennings, 1999]. Finally, a discussion on object oriented versus agent programming can be found in [Odell, 1999], since in this paper an agent system is constructed by partially using object oriented programming and object oriented design methods this is very relevant.

Another objective of our project is to learn from the customers, the most relevant field concerning this objective is web mining since it is focussed on finding patterns amongst visitors of websites. In [Mobasher *et al.*, 1996] a proposal for a framework for web mining is done, their work is aimed at finding association rules and sequential patterns in the data gathered from their visitors. The main difference in the approach used in this (and other) papers is that the data available for web mining is completely different from the data available within the MAGNET system. More on web mining can be found in for example [Wang *et al.*, 2000] and [Srivastava *et al.*, 2000]. An article more focussed on customer profiling is [Bounsaythip and Rintarunsala, 2001], the basic approach that should be taken when trying to find profiles of customers is discussed here. In our paper an approach similar to this one is taken. Mining through large amounts of data is computationally expensive, therefore finding an efficient datamining algorithm is becoming more and more necessary, in [Zaki *et al.*, 1996] sampling is shown to speed up the datamining process. In case the MAGNET system ever becomes a commercially used system this could be very relevant for us since large amounts of data become available for mining. When this is the case, the mining process could take too long with as a consequence that bids are not submitted in time. Finally, a successful application of building customer profiles and using these profiles can for example be seen at Amazon¹.

¹www.amazon.com

3 Early design

We have made a proposal for a more advanced design of the bidding process for the *supplier agent*. It is shown in figure 3 below.

As can be seen, the SUPPLIERAGENT contains the following components: COMMUNICATIONMANAGER, AGENTSTATE, BIDMANAGER, SALESANALYST, PRICEMANAGER and RESOURCEMANAGER.

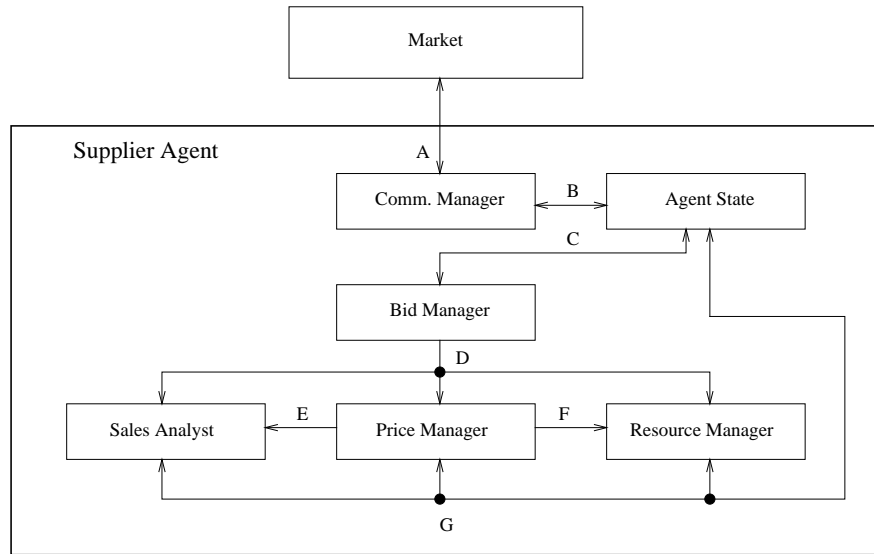
The communication with the outside world goes through the COMMUNICATIONMANAGER. This component communicates with the MARKET, this can be done in two ways: By means of direct communication, without interference of a server, or by means of communication through a server. The first one is mainly meant for experiments, the latter is consistent with the design shown in figure 2.

With the incoming communication, the AGENTSTATE is updated. This component maintains the current state of the agent and the market (which includes the current time on the market). All the components except the COMMUNICATIONMANAGER can subscribe to events with the AGENTSTATE component. When an event occurs, the AGENTSTATE notifies all subscribed components of this event.

When an RFQ arrives, the BIDMANAGER is notified by the AGENTSTATE component (given that the BIDMANAGER is subscribed to this event). The BIDMANAGER can then ask for the RFQ at the AGENTSTATE. If the BIDMANAGER needs market statistics, it issues a request with the AGENTSTATE. This component checks if it has got the statistics and if not it notifies the COMMUNICATIONMANAGER that it should retrieve them from the Market. When the COMMUNICATIONMANAGER receives the market statistics from the COMMUNICATION SERVICE, it updates the AGENTSTATE component. This component then notifies the BIDMANAGER of it. Now the BIDMANAGER can retrieve the market statistics from the AGENTSTATE. The BIDMANAGER can ask advice from the following components: SALESANALYST, PRICEMANAGER and RESOURCEMANAGER. Besides that, it determines whether to submit single or multiple bids. It also deals with problems like bidding on individual or blocks of tasks, are bids submitted as early as possible in the bidding cycle or do you wait until the last minute.

The SALESANALYST is concerned with giving advice on issues related to a particular customer.

The RESOURCEMANAGER is used for handling issues related to resources. This involves giving advice on issues related to resources. Further,



- A: Communication to and from the Market.
- B: Update the AGENTSTATE when an event occurs, and notify the COMMUNICATIONMANAGER of certain types of events and pass on information to it.
- C: Request for and passing of the market statistics. The RFQ is also retrieved from the AGENTSTATE and Bids are passed to the AGENTSTATE.
- D: The BIDMANAGER asks advice from the three components.
- E: What are my sales goals for these tasks?
- F: What will the resources cost for these tasks?
- G: Components can subscribe to certain events and be notified when one of these events occur. They also retrieve the information they need from the AGENTSTATE.

Figure 3: The bidding architecture

it is responsible for the execution of tasks that have been awarded.

The PRICEMANAGER determines the price of each task it wants to bid on. For this it needs to ask the SALESANALYST what the sales goal is for each particular task. It also needs to ask the RESOURCEMANAGER what the resource cost are for each of the tasks.

The last three components can ask for the information they need with the AGENTSTATE. They get their information similar to the way the BID-MANAGER gets it's information.

In order for the components to interact and to be able to work with different instances of these components, there needs to be some supervision by means of a tool. In the next section we investigate whether the Avalon tool is suitable to be used within the SUPPLIERAGENT, the tool is known to handle such interactions really well and easy. For a thorough explanation of this tool, see [Loritsch, 2001].

4 Investigation of the Avalon tool

In order for the components of the *supplier agent* to easily cooperate we need to have some tool for component management. Secondly, it should be easy to insert different types of the aforementioned components. We've examined the usefulness of the Avalon tool of the Apache organisation.

4.1 A brief introduction to Avalon

First let us give a brief introduction to the Avalon product. The idea for the Avalon project was to provide a framework to put together components and reuse code across a number of projects. Stefano Mazzocchi, Frederico Barbieri and Pierpaolo Fumagalli created the initial version, later on Berin Loritsch and Peter Donald joined the project. Nowadays Avalon is split up into five sub-projects, this has been done because the different pieces each have a different maturity level. The five sub-projects are:

Framework This is the basis of all the other projects and defines the interfaces and the default implementations for Avalon.

Excalibur Excalibur is a collection of server side components which you can use. Pooling implementations are also included in this, as well as database connection management and component management.

LogKit This is the logging tool used by the other sub-projects.

Phoenix Phoenix is the server kernel that manages the deployment and execution of services (it is implemented as server components called Blocks).

Cornerstone Avalon Cornerstone is a collection of Blocks (which include socket management and job scheduling) or services deployable in the Phoenix environment.

4.2 How Avalon works

Now an overview of how Avalon works is given. Avalon has been built with specific design principles, the two most important ones are treated, namely *Inversion of Control* and *Separation of Concerns*.

In *Inversion of Control* the component is always externally managed. The components lifecycle is controlled by the code that created the component.

Seperation of Concerns means that a component is looked upon from different viewpoints. Each viewpoint is a separate area of concern. Through research it was discovered that many concerns couldn't be addressed at class or even method granularity. The solution for this problem is implemented in Avalon by means of providing small interfaces that a component implements.

We are now ready to go more into implementation details. The first question we need to ask ourselves is how we can define components within the Avalon framework. In order to define a component in Avalon it is necessary to define a role for each component. There are a number of interfaces that can be extended in order to fit a component inside the Avalon framework. There are seven main interface categories: *Activity*, *Component*, *Configuration*, *Context*, *Logger*, *Parameters*, *Thread* and *Miscellany*. Each of those categories represents a unique concern area. Each component should at least implement one of these interfaces, but can also implement several of them.

The lifecycle of a component is split into three phases: *Initialization*, *Active Service* and *Destruction*. These phases are sequential in this order.

In Avalon it's possible to have single threaded as well as threadsafe components. Within the system there is also a possibility to use *Pooling*, if used it maintains a pool of Components and reuse instances. If this option is used, there's no need to worry about using threadsafe or single threaded, because this is dealt with when *Pooling* is used.

The general structure of the Avalon architecture contains the following elements: COMPONENTS, COMPONENTMANAGER, COMPONENTSELECTOR and COMPONENTCONTAINER. The COMPONENTS have already been treated. The COMPONENTMANAGER is responsible for managing the components, it can retrieve COMPONENTS by means of specifying the role. The COMPONENTSELECTOR is responsible for managing the instances of the COMPONENTS. In order to retrieve COMPONENTS it needs a specification of the role and an arbitrary object for a hint. Finally, the COMPONENTCONTAINER contains the COMPONENTS it is responsible for.

In order to ease testing for the MAGNET project, it would be nice to have a possibility to switch between components on the fly. Avalon is equipped with tools that makes this possible. It is also possible to configure the Avalon system by using XML files. With these XML files you can specify which components and which instances you want to have included in the Avalon System. The advantage of using this scheme is that you don't have to change

the code every time you want to use a different configuration.

4.3 Avalon and MAGNET

In figure 4 we have drawn a first proposal for the architecture using Avalon.

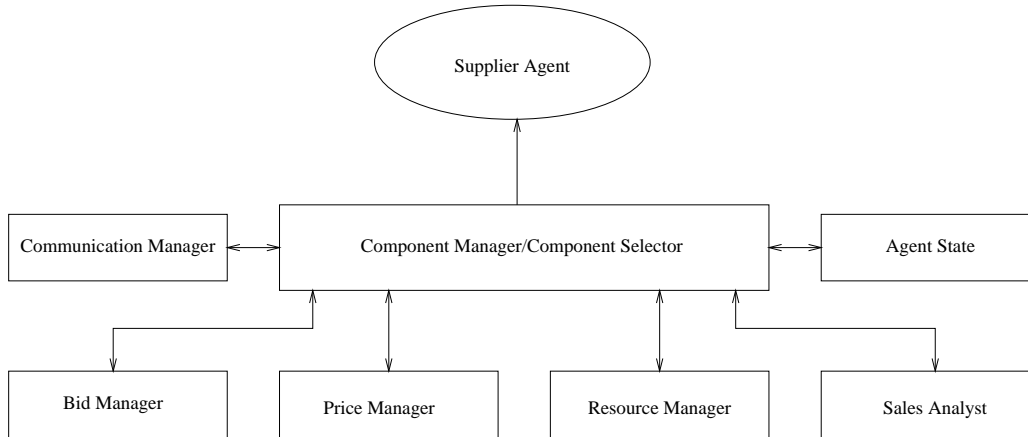


Figure 4: Proposed architecture using Avalon

As can be seen there are six components, namely `COMMUNICATION-MANAGER`, `AGENTSTATE`, `BIDMANAGER`, `SALESANALYST`, `PRICEMANAGER` and `RESOURCEMANAGER`. The `SUPPLIERAGENT` is modelled as a `COMPONENTCONTAINER`. This means it's **responsible** for the aforementioned components. The `SUPPLIERAGENT` basically has no other function but to control the lifecycle of all the components. All six components are **controlled** by the `COMPONENTMANAGER`. They all get a reference to the `COMPONENTMANAGER` in order to be able to access the other components. If there are multiple instances of a component, the `COMPONENTSELECTOR` can be retrieved from the `COMPONENTMANAGER` and with this the right instance can be selected.

4.4 Conclusion

Considering the given overview of the Avalon System, the Avalon system is suitable for regulating the control of the components specified in figure 4. A brief summary of the advantages of using Avalon: First of all, there are no problems in controlling the components. Secondly, it is easy to switch

between different instances of components on the fly. Third, by using XML files the managers can be configured without changing and compiling the code. Given these advantages the Avalon system is used for the component management in the *supplier agent*.

5 Designing the Supplier agent

Now that we have shown that the Avalon tool is usefull, we present a more detailed design in this section. A starting point for this design is the design proposed in figure 4. In the first section the design method that has been used is discussed.

5.1 Methodology

The first question which comes up when designing an agent is which design method to use. The method we are very familiar with is the one proposed by [Brazier *et al.*, 1998]. This method seems to be a good choice, since it is easy to specify all the information types and components. The actual framework consists of more than just a design method, it includes:

- a design method for compositional multi-agent systems.
- software tools to support system design
- a formal specification language for system design.
- an implementation generator to automatically translate specifications into code in specific implementation environments.
- verification tools for static properties of components such as consistency, correctness, completeness

Only the first two items have been used, since we only want to use the method to specify the general structure of the *supplier agent*. By using the software tools available for designing a multi-agent system using DESIRE it's easy to create graphics of how the components are arranged. The last three items haven't been used because there is no intention to implement the system using DESIRE since DESIRE is not meant to generate implementations, but can however be used to build a prototype.

Even if one is not familiar with the DESIRE method, it is very easy to comprehend. Before using DESIRE UML Use Cases have been drawn up to see which components are needed. After this, DESIRE is used and after that UML has been used to specify a more detailed design, including defining interfaces and information flow.

5.2 UML Use Cases

The use case diagram for the *supplier agent* is shown in figure 5. Now the use cases will be discussed. The Market is equivalent to the MAGNET server, but in this context the term MAGNET server is preferred.

Actors

- MAGNET Server
 - Stereotype:** Remote system
 - Communicates Links:**
 - to **UseCase** RFQ Status Update
 - to **UseCase** Bid Status Update
 - to **UseCase** Bid Award
- User
 - Stereotype:** User
 - Communicates Links:**
 - to **UseCase** Register
 - to **UseCase** Enter JobInfo
 - to **UseCase** Edit JobInfo
 - to **UseCase** Submit JobInfo
 - to **UseCase** Update JobInfo

System Boundary

- *supplier agent*
 - This is the *supplier agent* with it's User Interface.

Use Case Summary

- Register
- Login
- Enter JobInfo
- Edit JobInfo
- Submit JobInfo
- Update JobInfo

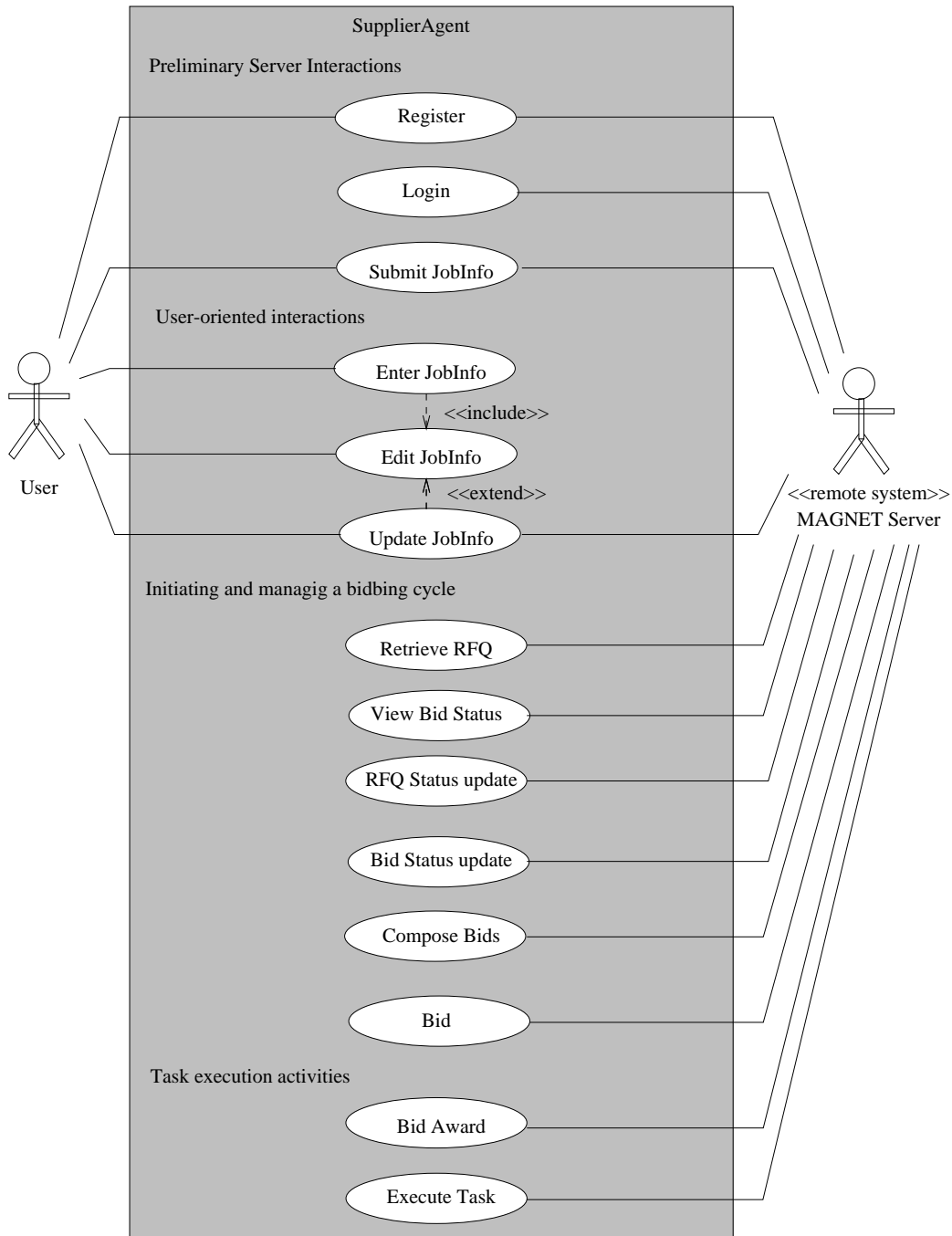


Figure 5: Use case diagram of the Supplier Agent

- Retrieve RFQ
- RFQ Status Update
- Compose Bids
- Bid
- View Bid Status
- Bid Status Update
- Bid Award
- Execute Task

Register

Goal *supplier agent* wants to register with the Server.

Summary Provide the *supplier agent* with identity to log in.

Trigger Agent initiated

Frequency Only once.

Preconditions

- SupplierAgent is not known to Server
- *supplier agent* may or may not have an identity

Normal Flow

1. User enters Identity data into *supplier agent*.
2. *supplier agent* makes Identity data persistent somehow.
3. *supplier agent* locates Server
4. *supplier agent* forwards Identity data to the Server.
5. Server validates Identity data.
6. If data is valid, Server allows *supplier agent* to log in, and persists this state.
7. Verify state and a security token is returned to *supplier agent*
8. *supplier agent* informs User of registration result.

PostConditions

- *supplier agent* has an Identity
- *supplier agent* is known to the Server, and the Server has validated the *supplier agent*'s Identity.

Login

Goal The *supplier agent* wants to log in to the Server.

Summary The *Login* UseCase is the sequence of steps in which the *supplier agent* logs in to the Server.

Trigger Agent initiated

Frequency At most three times during a session, or upon *supplier agent* restart. A session is the time between the arrival of an RFQ and the finish of the execution of the tasks in this RFQ.

Preconditions

- *supplier agent* is not connected to Server.
- *supplier agent* must be registered with the Server and have a valid security token.

Normal Flow

1. *supplier agent* locates Server.
2. *supplier agent* sends credentials to Server. Note that this is not a user login, but rather the *supplier agent* logging in to the Server. The *supplier agent* has its own identity, independent of any particular User. The issue of access control to the Agent itself is a separate issue.
3. Server validates *supplier agent*, and informs the *supplier agent* of the successfulness of the action.

PostConditions

- *supplier agent* is connected to Server.

Enter JobInfo

Goal The User makes the *supplier agent* aware of the JobInfo types it has.

Summary The User informs the *supplier agent* of the JobInfo types it has. There are three views of the JobInfo:

- The stored local version. This version is kept in the filesystem of the *supplier agent*. It's only accessible for the User and the *supplier agent*.
- The runtime representation maintained by the *supplier agent*.
- The runtime representation in the Server. With this version the Server knows what the JobInfo types of the *supplier agent* are and can therefore inform the *supplier agent* of the arrival of an appropriate RFQ.

Of each view, there is only one instance. The local view and the runtime representation maintained by the *supplier agent* should be identical, however not having the runtime representation would cause a severe performance decrease because of the reading and writing of files that has to be done everytime the information is needed. Not having a stored local version would be unsafe. The runtime representation in the Server may differ from the other views. This is done so the *supplier agent* won't be notified of RFQ's concerning JobInfo types it doesn't want to use at this moment.

Trigger User initiated

Frequency Every time a new JobInfo type is added.

Preconditions

- The JobInfo types are known to the Market.

Normal Flow

1. User requests to add a JobInfo type which has to be known to the Market.
2. *supplier agent* presents the JobInfo types which are known to the Market.
3. User enters JobInfo data. For each job info type, it contains one entry.

PostConditions

- The JobInfo types entered by the User are available to the *supplier agent*.

Edit JobInfo

Goal The entered JobInfo types can be edited.

Summary The User edits the JobInfo types.

Trigger User initiated

Frequency Every time a JobInfo type needs to be edited.

Preconditions

- JobInfo types have been entered.
- JobInfo types are known to the Market.

Normal Flow

1. The JobInfo types can be added, edited or deleted.

PostConditions

- User-initiated changes that are legal have been implemented.

Submit JobInfo

Goal The JobInfo types of which the *supplier agent* wants the Server to know about need to be submitted.

Summary This UseCase models the submitting of the JobInfo the *supplier agent* wants the Server to be informed about to the Server.

Trigger Agent initiated

Frequency Once.

Preconditions

- The *supplier agent* has to be logged in.
- The JobInfo types have been entered and are known to the Market.

Normal Flow

1. The *supplier agent* submits the JobInfo to the Server.
2. The Server checks if the format of the JobInfo type is correct and known.
3. The Server notifies the *supplier agent* whether the JobInfo is accepted.
4. The *supplier agent* notifies the User on whether the submit was successful.

PostConditions

- The submitted JobInfo is accepted by the Server. From this point the Server will notify the *supplier agent* when an RFQ arrives which is considered to be interesting according to the submitted JobInfo types.

Update JobInfo

Goal The JobInfo types the Server maintains of the *supplier agent* need to be updated.

Summary If the Server has already received JobInfo from this *supplier agent*, the *supplier agent* must be able to update them. This is modelled in this UseCase.

Trigger User initiated

Frequency Every time the JobInfo types change and the *supplier agent* wants the Server to know about it.

Preconditions

- The *supplier agent* has to be logged in.
- JobInfo types have been submitted to the Server.

Normal Flow

1. The User makes a set of legal changes to the JobInfo. Legal changes are for example adding or deleting certain types.
2. The *supplier agent* submits the JobInfo types to the Server.
3. The Server checks if the format of the JobInfo types is correct and known.

4. The Server notifies the *supplier agent* whether the JobInfo types are accepted.
5. The *supplier agent* notifies the User on whether the update was successful.

PostConditions

- The JobInfo types have been updated on the Server.

Retrieve RFQ

Goal The RFQ that has arrived at the Server must be passed on to the *supplier agent*. This UseCase is concerned with getting RFQ's.

Preconditions

- The *supplier agent* is logged in.
- The *supplier agent*'s JobInfo types have been submitted.
- An RFQ Status update has taken place.

Normal Flow

1. *supplier agent* asks the Server for the new arrived RFQ.
2. Server retrieves the RFQ.
3. Server returns the RFQ to the *supplier agent*.

PostConditions

- The RFQ for which a notification was sent by the Server has been retrieved.

RFQ Status Update

Goal The *supplier agent* is notified of every state change of an RFQ.

Summary The Server informs the *supplier agent* when a suitable RFQ arrives. RFQ's can arrive at any time to make it asynchronous. All the other state changes of this RFQ will be dealt with within the *supplier agent*. This is modeled within this UseCase.

Trigger Server initiated

Frequency Every time an RFQ which is suitable for this *supplier agent* arrives at the Server.

Preconditions

- The *supplier agent* is logged in.
- The *supplier agent* has submitted it's JobInfo

Normal Flow

1. Server has determined that an RFQ was posted that the *supplier agent* should be informed of immediately.
2. Server signals the *supplier agent* that the RFQ was posted.
3. The *supplier agent* will take some autonomous action to respond to the RFQ notification.

PostConditions

- The RFQ notification was passed to the *supplier agent*.

Compose Bids

Goal The collection of bids needs to be determined.

Summary This UseCase models the process of determining a collection of bids for a received RFQ.

Trigger Agent initiated

Frequency Once for every received RFQ.

Preconditions

- *supplier agent* has received an RFQ.

Normal Flow

1. The User or *supplier agent* can compose a bid, if the *supplier agent* determines what to bid then the next steps will be taken. Else the User will determine what to bid.
2. *supplier agent* collects market statistics.
3. *supplier agent* determines the amount of time available for the determination of the collection of bids.
4. *supplier agent* determines which tasks to bid on, looking at the client who issued the RFQ. This means that the *supplier agent* determines the tasks suitable for this client.
5. *supplier agent* determines which of the aforementioned tasks should be considered when taking the resources into account.
6. *supplier agent* determines the number of bids to submit.
7. *supplier agent* determines how to compose these bids.
8. *supplier agent* determines the time-window for the tasks included within the bid.
9. *supplier agent* determines the milestones it wants to include in the bid.
10. *supplier agent* determines the price for each of the bids.

Alternate Flow

- If the *supplier agent* determines what to bid and it chooses not to bid then no bid is composed.

PostConditions

- A collection of bids has been made for the RFQ.

Bid

Goal The bid needs to be communicated to the market.

Summary The *supplier agent* can issue a bid, this is modeled within this UseCase.

Trigger Agent initiated

Frequency Once for every bid within the collection.

Preconditions

- *supplier agent* is logged in.
- *supplier agent* has composed a collection of bids.

Normal Flow

1. *supplier agent* determines whether it's time to issue a specific bid.
2. *supplier agent* issues the bid when it's time to do so.

PostConditions

- The bid has been communicated to the market.

View Bid Status

Goal The status of a bid on a particular RFQ can be viewed.

Summary This UseCase models the retrieving of the status of a bid which has been issued concerning a particular RFQ.

Trigger Agent initiated

Frequence Every time the *supplier agent* wants to know the status of a bid it has done.

Preconditions

- The *supplier agent* is logged in.
- A bid has been done on a particular RFQ.

Normal Flow

1. *supplier agent* asks Server for the status of the bid.
2. *supplier agent* retrieves the status.

PostConditions

- The *supplier agent* has retrieved the Status of the bid.

Bid Status Update

Goal The *supplier agent* must be made aware of all the status changes of the bid it has done.

Summary This UseCase models the updating of the status of a submitted bid. There are seven status updates possible for a bid:

1. Bid being considered
2. Bid Awarded
3. Bid Not Awarded
4. Earliest time to start with a Task within the Bid
5. Latest time to start with a Task within the Bid
6. Earliest time to finish with a Task within the Bid
7. Latest time to finish with a Task within the Bid

Trigger Server and *supplier agent* initiated

Frequency $2 + (4 * \text{number of tasks within a bid})$ for each bid. This number is calculated as follows: for every bid exactly 1 notification can be sent that the bid is being considered. Secondly, the bid awarded or the bid not awarded occurs exactly one time per bid. Finally, all the other times can occur once for every task.

Preconditions

- The *supplier agent* is logged in.
- The *supplier agent* has submitted a bid.

Normal Flow

1. Server has determined that an event occurred concerning a bid issued by this *supplier agent*.
2. Server signals the *supplier agent* that an event occurred concerning this bid.

3. If the signal is *Bid being considered* the *supplier agent* takes some action, what is to be done (TBD).
4. If the signal is *Bid Awarded* the *supplier agent* will take the actions necessary to execute the tasks within the bid.
5. If the signal is *Bid Not Awarded* the *supplier agent* updates its resources.
6. If the signal is *Earliest time to start with a Task within the Bid* the *supplier agent* takes some action, what is TBD.
7. If the signal is *Latest time to start with a Task within the Bid* the *supplier agent* takes some action, what is TBD.
8. If the signal is *Earliest time to finish with a Task within the Bid* the *supplier agent* takes some action, what is TBD.
9. If the signal is *Latest time to finish with a Task within the Bid* the *supplier agent* takes some action, what is TBD.

PostConditions

- The *supplier agent* has been notified of the event concerning the issued bid and has taken the appropriate actions.

Bid Award

Goal The *supplier agent* must be able to handle actions related to the awarding and not-awarding of bids.

Summary This UseCase describes how a *supplier agent* takes the necessary actions when a bid has been awarded or not.

Trigger Agent initiated

Frequency Once every bid.

Preconditions

- A bid has been issued.
- The status of this bid is Awarded or Not Awarded.

Normal Flow

1. Agent determines whether the bid has been awarded.
2. If the status of the bid is *Bid Not Awarded* then the reservation of the resources will be released.
3. If the status of the bid is *Bid Awarded* then the resources will be reserved, how this is done is TBD.
4. If the bid is awarded the *supplier agent* checks if the deposit has been paid.
5. If the deposit has been received and while the late finish time of the bid has not been reached the *supplier agent* executes the tasks.
6. After each milestone the *supplier agent* will communicate the progress to the Server.
7. After all tasks have been executed or when the late finish time has arrived the *supplier agent* communicates the tasks it has finished to the Server.
8. The *supplier agent* checks if all payments have been done.
9. When all payments have been done the bid is marked as done.

Alternate Flow

- If the status of the bid is *Bid Awarded* but the deposit has not been paid yet, the *supplier agent* waits with the reservation of the resources and the execution of the tasks.
- If the status of the bid is *Bid Awarded* and the *supplier agent* has begun executing the tasks within the bid but it fails to complete one or more of them (either it isn't capable of completing the task, or the deadline of the bid has expired), then the *supplier agent* will pay a penalty for the tasks it could not complete.
- If the *supplier agent* is finished with all the tasks within the bid but it hasn't received the payment for it before the appropriate deadline, the *supplier agent* will add a penalty fee to the payment.

PostConditions

- The resources have been correctly updated.

Execute Task

Goal A *supplier agent* needs to be able to execute a task.

Summary This UseCase models the execution of a task within a bid which has been awarded.

Trigger Agent initiated

Frequence Once for every task within a bid.

Preconditions

- *supplier agent* is logged in.
- A bid has been awarded.
- The resources needed for this task have been reserved.

Normal Flow

1. The *supplier agent* determines the exact time to start with this task.
2. If it's time to execute the task, the agent uses the reserved resources for the time the task takes.
3. The task is executed untill either the deadline is reached or the task has been fully excuted.
4. When the task was fully executed, the *supplier agent* marks the task as executed.

Normal Flow

1. If the deadline of this task is reached before the task could be fully executed, the task is marked as not completed.

PostConditions

- The task has been marked as executed or as not completed.

5.3 Design using DESIRE

DESIRE specifications are based on architecture composed of components with a hierarchical relation between them. Each component has its own input and output information types and uses its own task control knowledge, so the system is structured in a decentralised manner. The structure within the components is hidden from the “outside world”, only the interface types are defined. A component can be composed from multiple sub-components if the task the component performs is too complex for one component to manage.

There are a number of aspects included when making a conceptual design using DESIRE. The most important ones are listed below:

1. Specifying the information types, the information types can be specified on multiple levels, namely object and meta-levels.
2. Specifying the components and their tasks.
3. Designing the components using a pre specified *Generic Agent Model*. This model contains the following components: OWN PROCESS CONTROL, AGENT INTERACTION MANAGEMENT, MAINTENANCE OF AGENT INFORMATION, WORLD INTERACTION MANAGEMENT, MAINTENANCE OF WORLD INFORMATION, AGENT SPECIFIC TASK and COOPERATION MANAGEMENT. We will not go into further detail, see [Brazier *et al.*, 1998] for more details.
4. Specifying the knowledge bases of the components.
5. Specifying the sequence of control and tasks.

We didn't use all these features, since not all of them are useful for us. Only item 1 and item 2 have been used. We didn't use item 3 because we are already restricted to the design shown in figure 3. The aspect mentioned at item 4 also isn't used because there are no knowledge bases yet. Finally the last item has been dealt with in the UML part.

5.3.1 Specifying the components

First of all, the components are specified from the multi-agent perspective. The components are specified in figure 6 below.

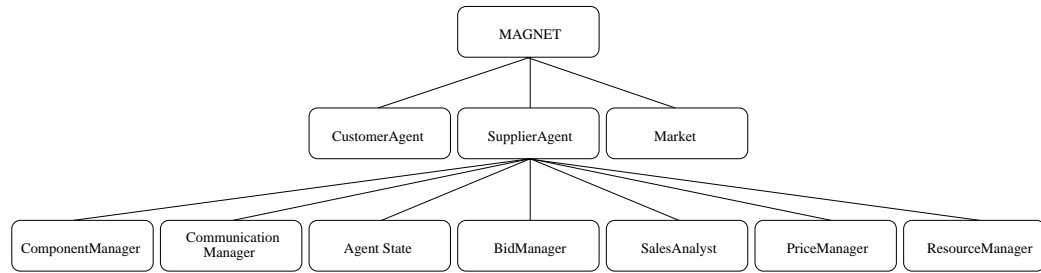


Figure 6: Multi-agent perspective

Rationale for the Multi-agent perspective The top-level is the MAGNET System. The system has been split up into 3 components on the second level: CUSTOMERAGENT, SUPPLIERAGENT and the MARKET. We've only focussed on the SUPPLIERAGENT part since the other components are outside the scope of this project. The SUPPLIERAGENT is split up into seven components, the COMPONENTMANAGER, COMMUNICATIONMANAGER, AGENTSTATE, BIDMANAGER, SALESANALYST, PRICEMANAGER and the RESOURCEMANAGER

The SUPPLIERAGENT is responsible for the components managed by the COMPONENTMANAGER.

The task of the COMPONENTMANAGER is to manage the components owned by the SUPPLIERAGENT. It can provide it's components with references to other components it manages. The COMPONENTMANAGER also includes the COMPONENTSELECTOR as mentioned in the description of the Avalon system. It is used to retrieve the correct instance of a component in case there are more instances.

The AGENTSTATE is responsible for maintaining the state of the agent and the market. This consists of keeping track of time and events coming from the market. It also includes the saving of all information that needs to be maintained, like bids that have been done and incoming RFQ's. Furthermore, components can subscribe to certain types of events and be notified by the AGENTSTATE when this event occurs.

The task of the COMMUNICATIONMANAGER is to receive information from and communicate information to the MARKET. Information that can be received from the MARKET is for example an RFQ or a bid award. Bids and penalty payment are examples of communications to the MARKET.

The task of the BIDMANAGER is to decide whether to submit single or

multiple bids, to bid on individual tasks or a block of tasks. Further more it decides if the bid is done early in the bidding cycle or wait until the last minute. Another issue dealt with in the `BIDMANAGER` is whether to bid your entire available time windows or restrict it. Finally, it deals with issues related to doing short-duration high cost or long-duration low cost bids.

The task of the `SALESANALYST` is to give other components advice when they want to take the customer into account.

The `RESOURCEMANAGER` is responsible for determining, given the current resources and the tasks, what tasks to bid on. It is also responsible for the management of the resources, this includes the execution of awarded tasks and handling the payments accompanying these tasks.

The task of the `PRICEMANAGER` is to determine the price to bid. It determines this price using the sales goals for these tasks, which it retrieves from the `SALESANALYST`. In order to determine the price, it also needs to know what the resources for these tasks cost. This information is retrieved from the `RESOURCEMANAGER`.

5.3.2 Defining the interfaces of the components

Now we are ready to examine the interfaces of the components. Let us first give a brief overview of how the system has been organized. For more information on information types, see the definitions of the interfaces.

In figure 7 the `COMPONENTMANAGER` is not shown, because it would make the figure less clear. Basically, all the `COMPONENTMANAGER` does is provide references to the other components. When the `SUPPLIERAGENT` becomes active first it will have to check whether it is registered or not, this is done within the `COMMUNICATIONMANAGER`. If it wasn't registered for this `MARKET` it will register itself. The registration is used to login to the `MARKET`. Now the `JobInfo` types will be checked and, if necessary, updated or submitted. This is done by the `RESOURCEMANAGER` or the user after which the `AGENTSTATE` is updated. The `AGENTSTATE` uses the `COMMUNICATIONMANAGER` to communicate the `JobInfo` to the `MARKET`. After this has been done the `COMMUNICATIONMANAGER` will be notified of interesting `RFQ`'s posted on the `MARKET`. When an interesting `RFQ` is posted the `COMMUNICATIONMANAGER` will be notified and retrieves the `RFQ` from the `MARKET`, then it updates the `AGENTSTATE` component. The `AGENTSTATE` component will then notify the components which have subscribed to the events about `RFQ`'s, that it is available. The `BIDMANAGER` retrieves the

RFQ from the AGENTSTATE. It can ask advice on what to do from the SALESANALYST, RESOURCEMANAGER and the PRICEMANAGER. If any of these components need market statistics to be able to determine their advice, they can ask the AGENTSTATE to get it for them. The AGENTSTATE checks if the information is already present and if not it will notify the COMMUNICATIONMANAGER that it needs to get this information from the MARKET. The components will know when the market statistics are available to them when the AGENTSTATE component notifies them. They can then get the market statistics from it. When the BIDMANAGER has received the advice it asked for, it makes a collection of bids and determines when a bid needs to be communicated to the MARKET. If it is time to issue a certain bid, the BIDMANAGER puts the bid in the AGENTSTATE and this component saves the bid and notifies the COMMUNICATIONMANAGER that the bid has to be communicated to the MARKET. When the bid award time has arrived the COMMUNICATIONMANAGER can receive an event that it's bid has been accepted. If this is true, the AGENTSTATE is updated. The AGENTSTATE will now notify all the components that subscribed to this event that a bid has been awarded. Now the RESOURCEMANAGER reserves the resources needed for these tasks. The RESOURCEMANAGER executes the tasks and when it is finished with it, the results will be put in the AGENTSTATE. This component uses the COMMUNICATIONMANAGER to communicate the results to the MARKET.

Components on the third level Now the interfaces of the components on the third level are described (see table 1).

Component Manager The COMPONENTMANAGER has one input information type: *Role*. This information type is needed to specify the role of the component the COMPONENTMANAGER needs to retrieve. The output information type is *Reference*. With this, the reference of the component to be retrieved is passed to the requesting component. Each component therefore has the *Reference* information type on the input side and the *Role* information type on the output side. When explaining the interfaces of the components we will not treat this any more.

Communication Manager The COMMUNICATIONMANAGER has the following input information types: *Event* and *Reference*. The *Event* input

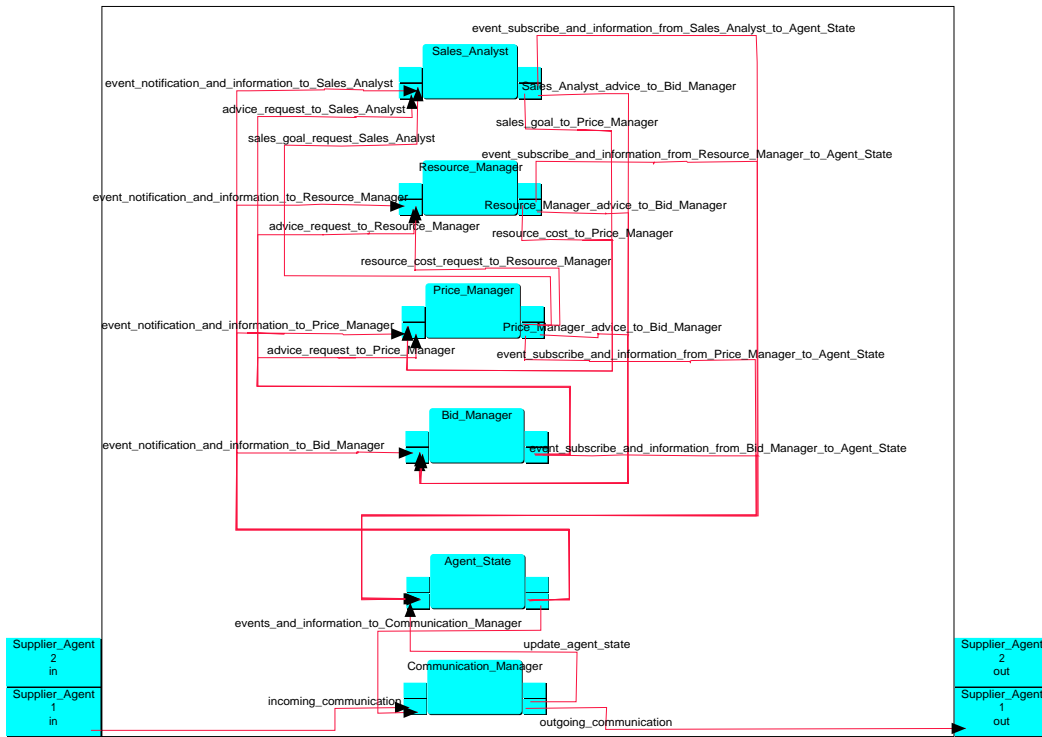


Figure 7: The architecture of the Component Manager

process	input information types	output information types
Component Manager	Role	Reference
Communication Manager	Event Reference	RFQ Collector Bid Request Role AgentTime Penalty Milestone BidCompleted Payment
Agent State	RFQ Collector Request AgentTime Bid Reference Penalty Milestone BidCompleted EventSubscription Payment	RFQ Collector Bid Event Role
Bid Manager	Event RFQ Collector TaskPlan Price Reference AgentTime	EventSubscription Bid Request TaskPlan Role AgentId
Sales Analyst	Collector RFQ Event AgentId Reference TaskPlan	TaskPlan SalesGoal EventSubscription Request Role Milestone
Resource Manager	Collector TaskPlan Reference Event	TaskPlan Price Role EventSubscription Request TimeWindow Penalty Milestone BidCompleted
Price Manager	Collector TaskPlan SalesGoal Price Event Reference AgentId	TaskPlan Price EventSubscription Request Role AgentId

Table 1: Specification of the third level interfaces

information type is used to receive information from the MARKET and the AGENTSTATE. The *Event* information type can contain one of the following information types: *RFQ*, *Collector*, *Bid*, *Request*, *AgentTime*, *Penalty*, *Milestone*, *BidCompleted* and *Payment*.

The following output information types are used by the COMMUNICATIONMANAGER: *RFQ*, *Collector*, *Bid*, *Request*, *AgentTime*, *Penalty*, *Milestone*, *BidCompleted* and *Payment*. The *RFQ* is needed as an output information type to pass through the RFQ to other components. The market statistics are passed through to other components with the *Collector* information type. The bid that has been made is passed on with the *Bid* information type. We use the information type *Request* as the information type for doing requests from the MARKET. The current time on the market is passed through with the information type *AgentTime*. When a SUPPLIERAGENT is not able to finish the tasks it is supposed to execute within the appropriate time, a penalty has to be paid, for this the **Penalty** information type is used. Milestones that need to be communicated to the MARKET are communicated with the information type **Milestone**. If the last task of the bid that was awarded is completed, the information type *BidCompleted* is used to communicate to the MARKET. Finally, the *Payment* information type is used to pass information from the MARKET concerning payments.

Agent State The AGENTSTATE component has the following input information types: *RFQ*, *Collector*, *Request*, *AgentTime*, *Bid*, *Penalty*, *Milestone*, *BidCompleted*, *EventSubscription* and *Payment*. The *RFQ* information type is used to receive RFQ's from the COMMUNICATIONMANAGER. To be able to receive statistics retrieved from the MARKET by the COMMUNICATIONMANAGER the *Collector* information type is included. Components can issue requests to the AGENTSTATE component, for this the *Request* information types is used. Since the SUPPLIERAGENT needs to be synchronized with other agents on the MARKET. The time within the AGENTSTATE component must be set to the aforementioned value, for this *AgentTime* is an information type on the input side. The *Bid* information type is used to be able to issue bids (this goes through the AGENTSTATE component because the database within this component needs to be updated). When a penalty is put on the SUPPLIERAGENT or when the SUPPLIERAGENT puts a penalty on another agent the *Penalty* information type is used. While executing tasks, milestones have to be communicated, this is done through the

Milestone information type. When the tasks are fully executed, or when it is the latest finish time, the completed tasks are communicated by using the *BidCompleted* information type. The *EventSubscription* information type is needed for other components to be able to subscribe to certain types of events. Finally, the SUPPLIERAGENT should be able to receive payments, for this the *Payment* information type is used.

AGENTSTATE has the following output information types: *RFQ*, *Collector*, *Bid*, *Event* and *Role*. The *RFQ* information type is used to provide the BIDMANAGER with the RFQ's so it can begin the constructing of a bid. Statistics that have been requested can be provided through the *Collector* information type. Components might want to use old bids to improve their performance on their task, these bids can be passed through to them by using the *Bid* information type. Events that include a certain type of information can be passed through to the components that have subscribed to this event, for this the *Event* information type is included.

Bid Manager The BIDMANAGER has the following input information types: *Event*, *RFQ*, *Collector*, *TaskPlan*, *Price* and *Reference*. The *Event* information type is used to pass events containing a certain type of information through to this component. Information on RFQ's is passed through using the *RFQ* information type. Statistics that have been requested are passed to this component with the *Collector* information type. The advice about what tasks to bid on are passed using the *TaskPlan* information type. The *Price* information type is used to pass through advice concerning the price.

The output information types of the BIDMANAGER are: *EventSubscription*, *Bid*, *Request*, *TaskPlan*, *Role*, *RFQ* and *AgentId*. With the *EventSubscription* information type the BIDMANAGER can subscribe to certain types of events. The *Bid* information type is used to pass the bid that needs to be issued. Requests can be done with the information type *Request*. The BIDMANAGER can ask advice about tasks by using the *TaskPlan* information type. The *RFQ* information type is used to ask advice about what tasks to bid on concerning the salesgoals. Finally, the *AgentId* information type is passed to identify the agent the BIDMANAGER wants information about.

Sales Analyst The SALESANALYST has the following input information types: *Collector*, *RFQ*, *Event*, *AgentId*, *Reference* and *TaskPlan*. The *Collector* information type is used to pass the requested statistics. Informa-

tion about RFQ's is passed with the *RFQ* information type. With the *Event* information type certain types of information are passed to the SALESANALYST. The identity of the agent which is being discussed is passed on with the *AgentId* information type. The *TaskPlan* information type is used to pass information about the tasks to be considered.

The output information types of the SALESANALYST are: *TaskPlan*, *SalesGoal*, *EventSubscription*, *Request*, *Role*, *Role* and *Milestone*. The *TaskPlan* information type is used to pass information about the tasks to be considered. The *SalesGoal* information type contains information about the sales goals of the agent of which information was requested. The *EventSubscription* information type is used by the SALESANALYST to subscribe to certain types of events. Requests can be done by using the *Request* information type. Information on milestones are passed with the *Milestone* information type.

Resource Manager The RESOURCEMANAGER has four input information types: *Collector*, *TaskPlan*, *Reference* and *Event*. The *Collector* information type is used to pass the requested statistics. With the *TaskPlan* information type the tasks about which advice is requested are passed. Certain types of information are passed with the *Event* information type.

The output information types are: *TaskPlan*, *Price*, *Role*, *EventSubscription*, *Request*, *Penalty*, *Milestone*, *BidCompleted* and *TimeWindow*. The tasks about which advice is requested are passed using the *TaskPlan* information type. The resource cost are passed with the *Price* information type. The RESOURCEMANAGER can subscribe to certain type of events with the *EventSubscription* information type. The RESOURCEMANAGER can issue requests using the *Request* information type. If the RESOURCEMANAGER wants to issue a penalty it uses the *Penalty* information type. Milestones are passed with the *Milestone* information type. When the tasks that were awarded are completed, or when the latest time to finish has arrived, the RESOURCEMANAGER uses the *BidCompleted* information type to inform which tasks were completed. The *TimeWindow* information type is used to pass advice about which time windows to include.

Price Manager The input information types of the PRICEMANAGER are *Collector*, *TaskPlan*, *SalesGoal*, *Price*, *Event*, *Reference* and *AgentId*. The requested statistics are passed with the *Collector* information type. The

TaskPlan is used to receive a taskplan for which advice is requested. The requested sales goals are passed with the *SalesGoal* information type. The *Price* information type is used to pass information about the resource costs. The *Event* information type is used to pass certain types of information. Agent ID's are passed with the *AgentId* information type.

The PRICEMANAGER has the following output information types: *TaskPlan*, *Price*, *EventSubscription*, *Request*, *Role*, *AgentId*. The *TaskPlan* information type is used to pass the taskplan about which advice is requested. The advice on what price to bid is passed with the *Price* information type. The PRICE MANAGER can subscribe to certain types of events with the *EventSubscription* information type. Requests can be done using the *Request* information type. The *AgentId* information type is used to inform the SALESANALYST on the identity of the agent it wants advice about.

5.3.3 Information Exchange

In this section the information exchange on the third level is described.

In table 2 the links with the accompanying components and information types are displayed. All links are always on the object level because the information types used in the links are not statements about statements. Within the components there are meta-level information types, but the scope of this design does not include the design of the specific components. In the table the links concerning the COMPONENTMANAGER are not specified, since the links are all the same. From the COMPONENTMANAGER a link exists to every other component. These are used to pass references from other components. From each of the aforementioned components a link exists to the COMPONENTMANAGER in order to specify the role of the component it would like to have the reference from. We will not go into further details about the other links since the explanation would be trivial after the previous section.

information link	from process	output	to process	input
incoming communication	Market	Event	Communication Manager	Event
event and information to Communication Manager	Agent State	Event	Communication Manager	Event
update agent state	Communication Manager	RFQ, Collector, Agent-Time, Payment	Agent State	RFQ, Collector, Agent-Time, Payment
event subscription and information from Sales Analyst to Agent State	Sales Analyst	EventSubscription, Request	Agent State	EventSubscription, Request
event subscription and information from Resource Manager to Agent State	Resource Manager	EventSubscription, Request, Penalty, Milestone, BidCompleted	Agent State	EventSubscription, Request, Penalty, Milestone, BidCompleted
event subscription and information from Price Manager to Agent State	Price Manager	EventSubscription, Request	Agent State	EventSubscription, Request
event subscription and information from Bid Manager to Agent State	Bid Manager	EventSubscription, Request, Bid	Agent State	EventSubscription, Request, Bid
event notification and information to Sales Analyst	Agent State	Event, Collector	Sales Analyst	Event, Collector
event notification and information to Resource Manager	Agent State	Event, Collector	Resource Manager	Event, Collector
event notification and information to Price Manager	Agent State	Event, Collector	Price Manager	Event, Collector
event notification and information to Bid Manager	Agent State	Event, Collector, RFQ	Bid Manager	Event, Collector, RFQ
advice request to Sales Analyst	Bid Manager	RFQ, TaskPlan	Sales Analyst	RFQ, TaskPlan
sales goal request to Sales Analyst	Price Manager	AgentId	Sales Analyst	AgentId
Sales Analyst advice to Bid Manager	Sales Analyst	TaskPlan, Milestone	Bid Manager	TaskPlan, Milestone
sales goal to Price Manager	Sales Analyst	SalesGoal	Price Manager	SalesGoal
advice request to Resource Manager	Bid Manager	TaskPlan	Resource Manager	TaskPlan
resource cost request to Resource Manager	Price Manager	TaskPlan	Resource Manager	TaskPlan
Resource Manager advice to Bid Manager	Resource Manager	TaskPlan, TimeWindow	Bid Manager	TaskPlan, TimeWindow
resource cost to Price Manager	Resource Manager	Price	Price Manager	Price

to be continued on next page

<i>continuation of the previous page</i>				
advice request to Price Manager	Bid Manager	TaskPlan, AgentId	Price Manager	TaskPlan, AgentId
Price Manager advice to Bid Manager	Price Manager	Price	Bid Manager	Price
outgoing communication	Communication Manager	Bid, Request, Penalty, Milestone, BidCompleted	Market	Bid, Request, Penalty, Milestone, BidCompleted

Table 2: Specification of the third level links

This concludes our explanation of the components and their information exchange.

5.3.4 Knowledge composition, information types

In this section the information types are defined. Within the DESIRE design method this means defining the structure of the information types flowing between components. Ofcourse most of them have already been defined within the MAGNET system. We will treat these briefly. Let us first sum up all information types:

1. *RFQ*
2. *Bid*
3. *TaskPlan*
4. *SalesGoal*
5. *Price*
6. *Collector*
7. *Event*
8. *EventSubscription*
9. *Request*
10. *AgentTime*
11. *Role*
12. *Penalty*
13. *Milestone*
14. *BidCompleted*

15. *Payment*
16. *AgentId*
17. *TimeWindow*

RFQ The *RFQ* information type is used to represent the request for quotes, these are the messages used to issue a request of tasks that need to be completed. It is shown in figure 8 and is split up in three elements: *RFQDates*, *ProtocolElement* and *TaskPlan*. The *RFQDates* element is built up from the following relations: *BidDeadline*, *EarliestConsider* and *EarliestOffer*. All relations use the sort *DATE*. The *BidDeadline* represents the date before which a bid should be done on one or more of the subtasks of this RFQ. This is necessary to know because a supplier agent needs to know if a valid bid can still be done. Secondly, the *EarliestConsider* models the time by which the customer will begin considering the bids. The earliest time for a bid to be placed is modelled in the *EarliestOffer* relation. The *ProtocolElement* is treated in the next paragraph. The *TaskPlan* is discussed further on.

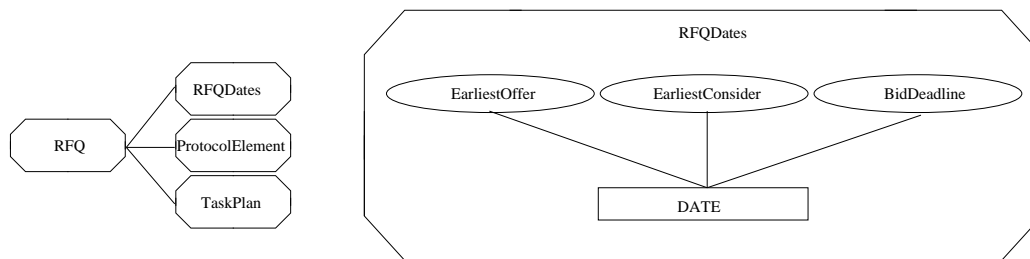


Figure 8: Information type: RFQ

ProtocolElement The *ProtocolElement* is used as a general information type for representing protocol elements (like the *RFQ* information type). The information type is shown in figure 9. It consists of the following relations: *AgentID*, *ID* and *SessionID*. *AgentID* is used to identify the agent of which this protocol element originates. The *ID* is used to identify the protocol element. Finally, the *SessionID* models the session to which this piece of protocol belongs.

TaskPlan The *TaskPlan* information type models a set of *SubTaskRequests* embedded in a precedence network, and is shown in figure 10. It

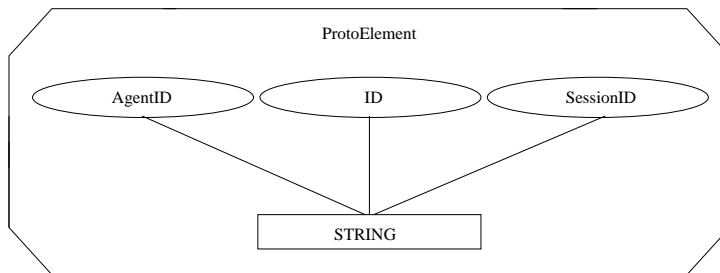


Figure 9: Information type: ProtocolElement

contains the following elements: It is split up in *TaskPlanInfo* and *SubtaskRequest*. The *TaskPlanInfo* contains the relations *PlanName*, *PlanDeadline* and *PlanStartDate*. The first relation uses the sort *STRING*. The *PlanDeadline* is used to denote what the latest deadline of all the tasks in this task plan is. The *PlanName* is the name to identify this task plan. The earliest *EarlyStartDate* of all tasks in the task plan is identified by the *PlanStartDate*.

SubtaskRequest The *SubtaskRequest* element is shown in figure 11 and contains the following relations: *Deadline*, *EarliestTimeToStart*, *ID*, *JobInfo*, *JobInfoName*, *TaskCommands* and *TaskName*. The relations *Deadline* and *EarliestTimeToStart* use the sort *DATE*. *JobInfo* contains information about the average costs, average duration, etc, about a job. Since the information type consists of a lot of information types and sorts and it had already been designed a more detailed view has been omitted, see the MAGNET Javadoc for more details. The other relations use the sort *STRING*. The *Deadline* is the deadline for the *SubtaskRequest*, *EarliestTimeToStart* is the earliest time to start executing the task within this *SubtaskRequest*. The id of the *SubtaskRequest* is specified in the *ID* relation. The relation *JobInfoName* specifies the name of the *JobInfo* contained in the *SubtaskRequest*. The *TaskCommands* relation contains the user-defined task commands. The *TaskName* specifies the name of the task within the *SubtaskRequest*.

SalesGoal The *SalesGoal* information type is used to model a sales goal and has one sort, namely *GOAL*. The sort *GOAL* consists of objects that are still to be determined (see section 8). The goal the *SUPPLIERAGENT* means to reach with a particular bid is modeled in this information type. This can

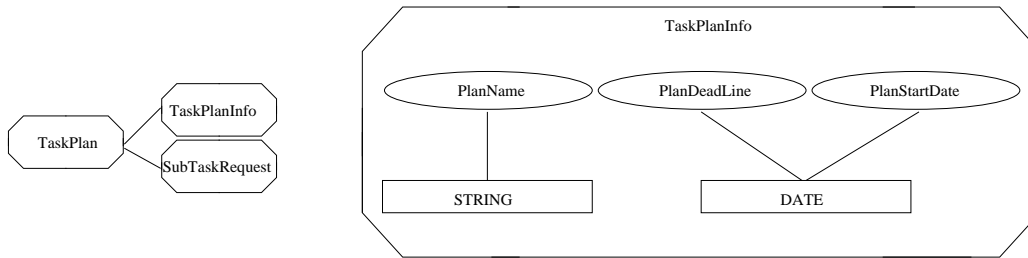


Figure 10: Information type: TaskPlan

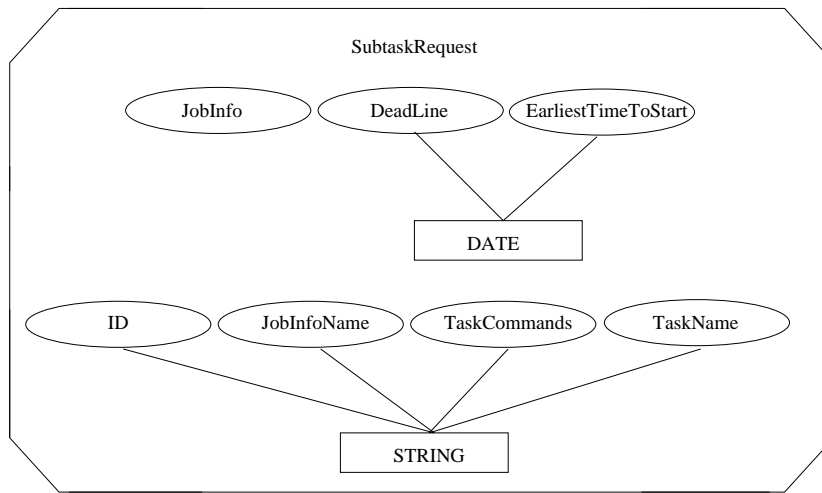


Figure 11: Information type: SubtaskRequest

be seen in figure 12.

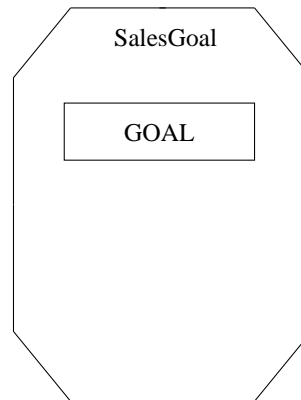


Figure 12: Information type: SalesGoal

Price This information type is used to model a price and contains the sort *LONG* (an integer value). This sort is used to denote the price of the resource costs. See figure 13.

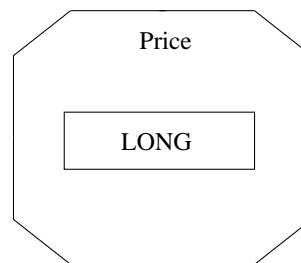


Figure 13: Information type: Price

Collector This information type consists of the following elements: *DataCollector*, *DiscountDataCollector*, *StorageDataCollector* and *TimedDataCollector*. This is shown in figure 14. This information type is used to model the market statistics. These elements are all used to model the market statistics. We will not go into further detail because it has not been specified yet.

Bid This information type consists of *BidData* and *Milestone* and is shown in figure 15. They are treated in more detail in the next paragraphs.

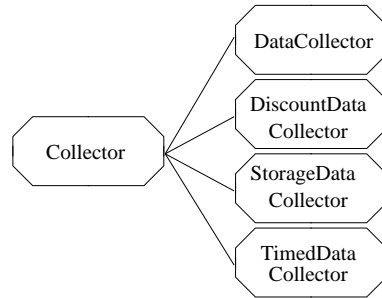


Figure 14: Information type: Collector

BidData This information type is shown in figure 15 and consists of the following relations: *Duration*, *EarlyStart*, *LateStart*, *EarlyFinish*, *LateFinish*, *LatestAccept*, *Cost* and *TaskSet*. The *Duration* relation describes the duration of the entire bid or of one task in the bid in particular. It uses the sort LONG. With the *EarlyStart* relation the earliest time to start with the execution of the tasks within the bid or of one task in the bid in particular. The sort used by this relation is DATE. The *LateStart* uses the DATE sort and denotes the latest time to start with the execution of the tasks within the bid or of one specific task in the bid. For the denotation of the earliest time to finish with the execution of the tasks within a bid or one task specific the *EarlyFinish* relation is used, this relation also uses the DATE sort. The latest time to finish with the execution of the tasks within the bid or one specific task within the bid is modelled with the *LateFinish*. For this it uses the sort DATE. The *LatestAccept* relation is used to denote the latest time on which the SUPPLIERAGENT can be notified of the acceptance of it's bid. For the cost for the execution of the tasks within the bid the *Cost* relation is used. This relation uses the sort LONG. The *TaskSet* relation specifies the set of taskID's in this bid, for this it uses the sort SET. The sort SET is a number of unordered taskID's.

Milestone The information type is shown in figure 16 and consists of two relations: *TaskId* and *MilestoneId*. The *TaskId* relation is used to represent the task for which this is a milestone. The sort used by *TaskId* is STRING. Each milestone within a bid has a unique id, for which the *MilestoneId* relation is used. This relation also uses the sort STRING.

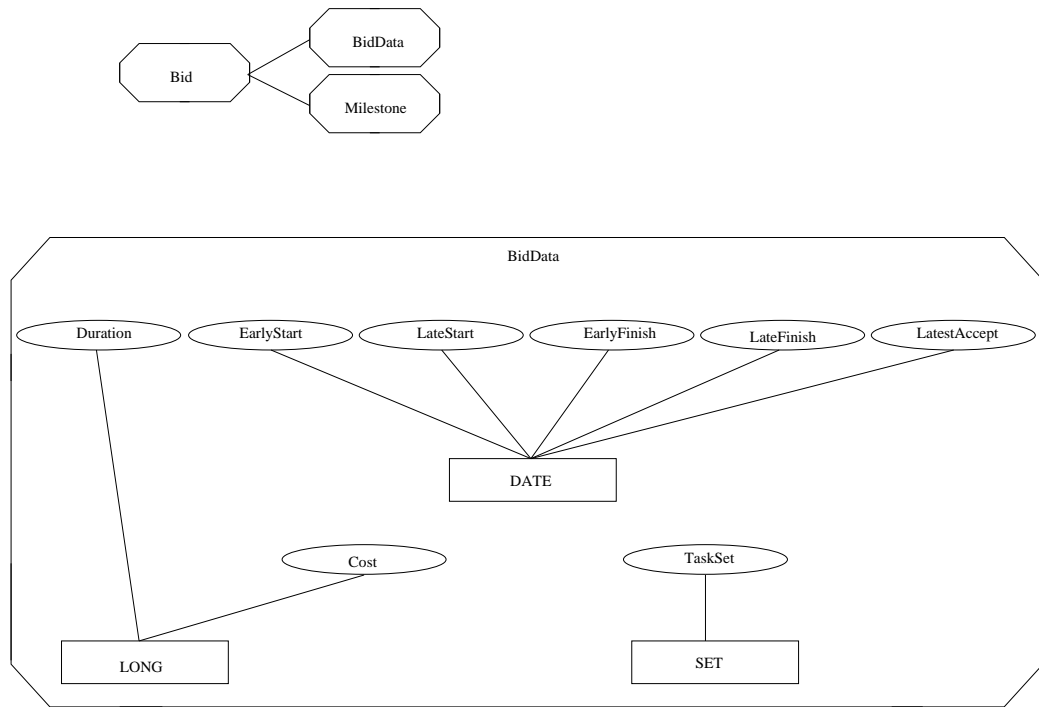


Figure 15: Information type: Bid

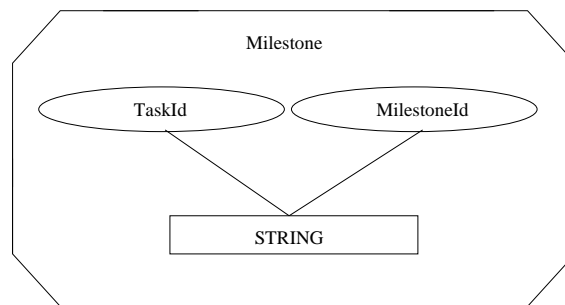


Figure 16: Information type: Milestone

Event The *Event* information type is shown in figure 17 and is used to pass events to the subscribed components. It consists of the following relations: *RFQArrivalEvent*, *RFQBidDeadlineEvent*, *RFQEarliestConsiderTimeEvent*, *RFQEarliestOfferEvent*, *BidDepositPaymentEvent*, *BidAwardTimeEvent*, *BidAwardEvent*, *BidEarliestFinishTimeEvent*, *BidEarliestStartTimeEvent*, *BidLatestStartTimeEvent*, *BidLatestFinishTimeEvent*, *BidPaymentEvent*, *BidPenaltyPaymentEvent*, *BidPenaltyNotificationEvent*, *StatisticsType1Event* and *StatisticsType2Event*. The relations *RFQArrivalEvent*, *RFQBidDeadlineEvent*, *RFQEarliestConsiderTimeEvent* and *RFQEarliestOfferEvent* all use the sort STRING. The sort is used by the relations to identify the RFQ on which the event is about. The aforementioned relations are used to model the events concerning a particular RFQ. The *RFQArrivalEvent* models the arrival of a new RFQ on the MARKET. When the deadline for the issuing of bids is reached, the *RFQBidDeadlineEvent* is used. The *RFQEarliestConsiderTimeEvent* denotes the earliest time on which the bids issued on this RFQ are considered. The earliest possible time on which a bid can be issued is passed by means of the *RFQEarliestConsiderTimeEvent*. To model events concerning bids, the following relations are used: *BidDepositPaymentEvent*, *BidAwardTimeEvent*, *BidAwardEvent*, *BidEarliestFinishTimeEvent*, *BidEarliestStartTimeEvent*, *BidLatestStartTimeEvent*, *BidLatestFinishTimeEvent*, *BidPaymentEvent*, *BidPenaltyPaymentEvent* and *BidPenaltyNotificationEvent*. All the relations use the sort STRING to identify the particular bid this event is about. The *BidDepositPaymentEvent* is used to model the payment of the deposit for this specific bid. Besides the aforementioned sort it also uses the sort LONG to model the amount of money paid. For each bid that has been issued there is a particular time on which the awarding of the bid takes place, this is modeled with the *BidAwardTimeEvent* relation. To model the award of a bid that has been issued the relation *BidAwardEvent* is used. Bids have an earliest finish time, earliest start time, latest start time and latest finish time to model these time event the relations used are, respectively, *BidEarliestFinishTimeEvent*, *BidEarliestStartTimeEvent*, *BidLatestStartTimeEvent* and *BidLatestFinishTimeEvent*. When a bid has been executed, a payment should be done. To model the arrival of a payment, the *BidPaymentEvent* relation is used. This relation uses the sort LONG to model the amount of money that has been paid. The *BidPenaltyPaymentEvent* denotes the payment of a penalty that has been put on the *customer agent*, for this the sort LONG is also used to model the amount paid. Finally, the *BidPenaltyNotificationEvent* is used to model an event of a penalty being put on the

SUPPLIERAGENT. To model the arrival of a certain type of statistics, the relations *StatisticsType1Event* and *StatisticsType2Event* are used. Both relations use the sort **STRING** to be able to identify the proper statistics so it can be retrieved from the **AGENTSTATE**.

When there can be multiple markets, there should be a composite key which can be used to identify RFQ's and bids. This is because there is no guarantee that the identification of the RFQ is unique on multiple markets.

EventSubscription The information type is shown in figure 18 and consists of one relation, namely *Subscribe*. The relation denotes the subscription to a certain type of event. It uses the sorts **EVENT_TYPE**, **STRING** and **OBJECT**. The **EVENT_TYPE** sort consists of objects that represent the types of events a component can subscribe to. With the **EVENT_TYPE** sort the type of the event one wants to subscribe to is passed. To be able to know what component wants to subscribe to this event, the **STRING** and **OBJECT** sorts are used. The **STRING** is used to denote the role of the component that wants to subscribe. The sort **OBJECT** is used to describe which instance of this component wants to subscribe.

Request In figure 19 this information type is shown. This information type contains one relation which is *DoRequest*. The relation is used to model the request that can be done for a certain type of information. The relation uses the sorts **REQUEST_TYPE**, **ROLE** and **HINT**. The **REQUEST_TYPE** sort consists of objects that represent a certain request type. The **STRING** and the **OBJECT** sorts are used for the same reason as explained in the *EventSubscription* information type.

AgentTime In figure 20 the *AgentTime* information type is shown. As can be seen, it contains three relations: *Date*, *Rate* and *Offset*. The relation *Date* is used to maintain the time, it uses the sort **DATE**. The rate at which the time is passing can be specified with the *Rate* relation, this relation uses the sort **DOUBLE**. With the relation *Offset* the offset value from the base is modeled. The sort **LONG** is used by this relation. The relations *Rate* and *Offset* are usefull for running experiments.

AgentId In figure 21 the information type *AgentId* is represented. It has the relation *Id* to model the identity of an agent and for this the sort **STRING**

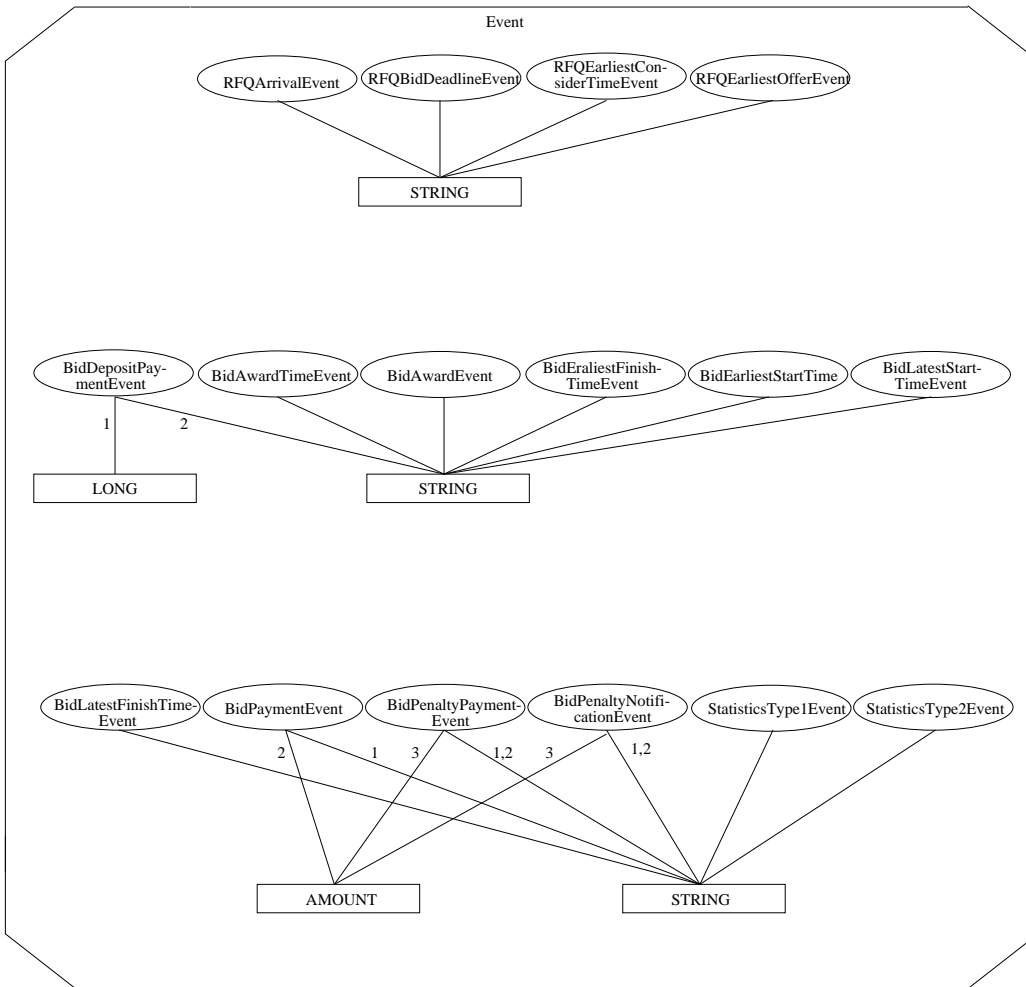


Figure 17: Information type: Event

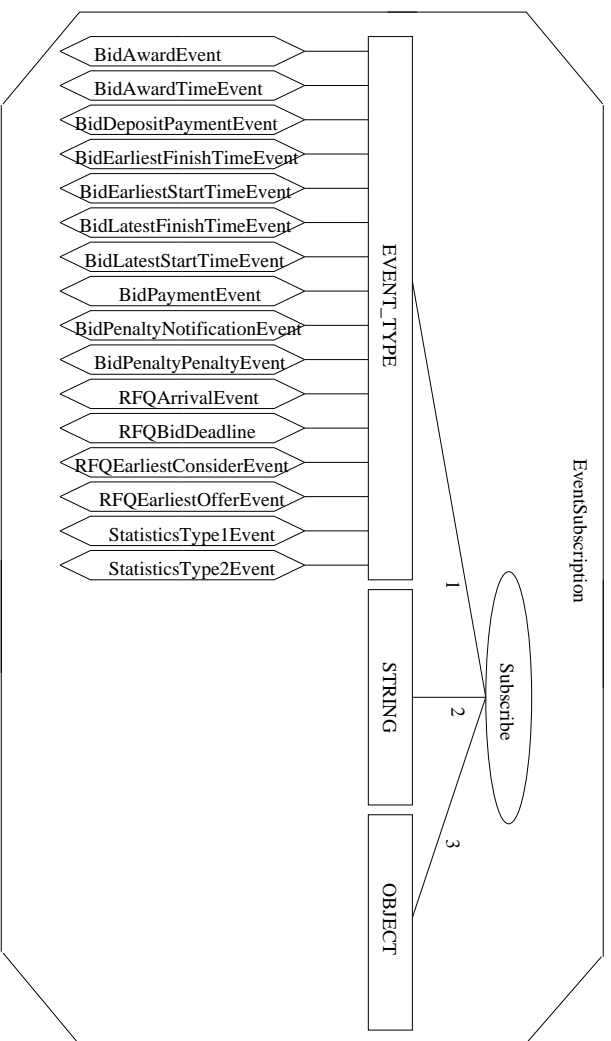


Figure 18: Information type: EventSubscription

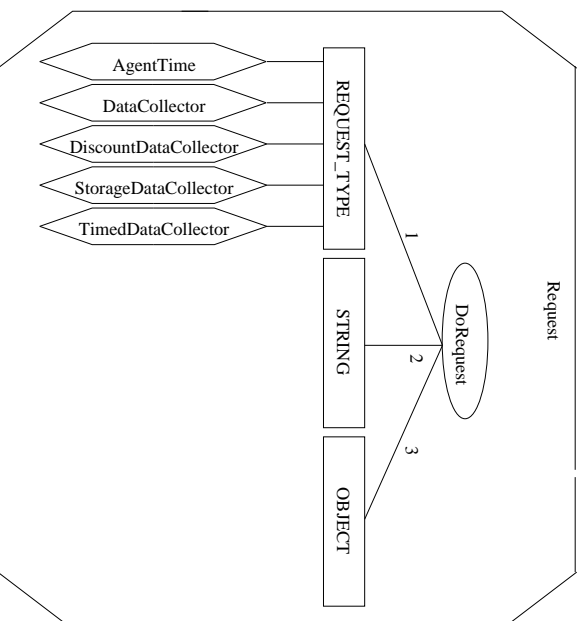


Figure 19: Information type: Request

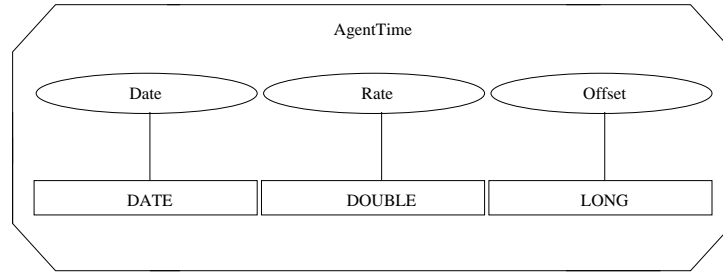


Figure 20: Information type: AgentTime

is used.

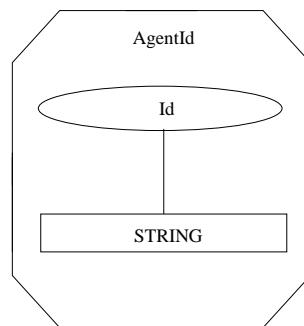


Figure 21: Information type: AgentId

Penalty The information type *Penalty* has one relation, namely *IssuePenalty*. With this relation the penalty that should be paid is modeled. The sort STRING is used to identify the agent which should pay the penalty. The sort STRING is also used to specify the bid for which the penalty should be paid. The last sort that is used is the sort LONG. This sort specifies the amount that should be paid as a penalty and is modeled in the information type *Event*. Figure 22 describes the *Penalty* information type.

Payment In figure 23 the *Payment* information type is shown. *Pay* is the only relation within this information type. The payment of a certain amount of money is modeled with this relation. It uses the sort PAYMENT_TYPE with which the type of payment is modeled. There are three types of payment which are represented as objects: *Deposit*, *Penalty* and *FinalPayment*.

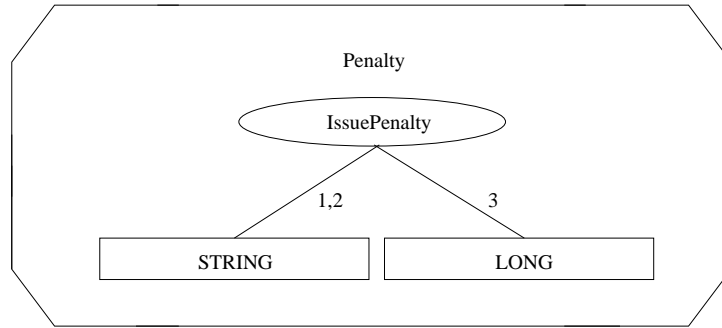


Figure 22: Information type: Penalty

Another sort used by the relation is `STRING` which is used to identify the agent from which the payment originates and to be able to identify the bid for which this payment is done. The sort is treated in the *Bid* information type. The sort `LONG` is used to denote the amount that is paid.

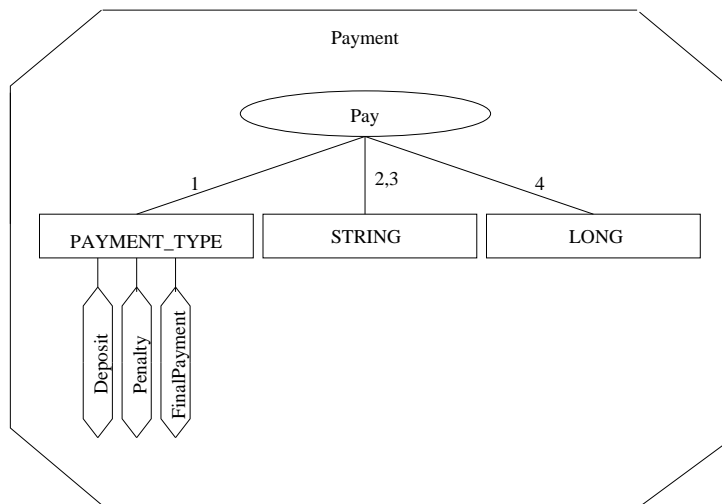


Figure 23: Information type: Payment

BidCompleted The *BidCompleted* information type consists of two information types: *Beliefs*, *Truth Indication* and *BidCompletedMetaInfo* and is shown in figure 24. The *Beliefs* information type is defined as a generic information type within DESIRE and is used to maintain information on

the world and other agents. The information type consists of the relation *belief*, using the sort INFO_ELEMENT and SIGN. The *Truth Indication* is also shown in the same figure and is used to add a sign to the relation. It contains the sort SIGN, which consists of the objects *pos* and *neg*. Finally, the *BidCompletedMetaInfo* consists of the sort INFO_ELEMENT using meta descriptions on the *BidCompletedInfo*. *BidCompletedInfo* uses the *TaskPlan* information type, which has already been treated, and the *BidInfo* information type consists of the following relations: *CustomerAgentId* and *BidId*. The *CustomerAgentId* relation is used to model the identity of the agent for which the information is meant. This relation uses the sort STRING to identify this agent. Another relation within the *BidCompletedInfo* is *BidId* which uses the sort STRING to identify the bid. The relation specifies the bid which this information is about. Since the relation is complicated if one is not familiar with DESIRE, let's give an example of an atom is

```
belief((TaskPlan, (BidId(bid1), AgentId(agent1))),pos)
```

where TaskPlan should be formed according to the definition of information type in figure 10 however, this would make the example less clear. The example indicates a bid with identification 'bid1' and from agent 'agent1' where there is a positive belief that all tasks included in the TaskPlan have been successfully completed.

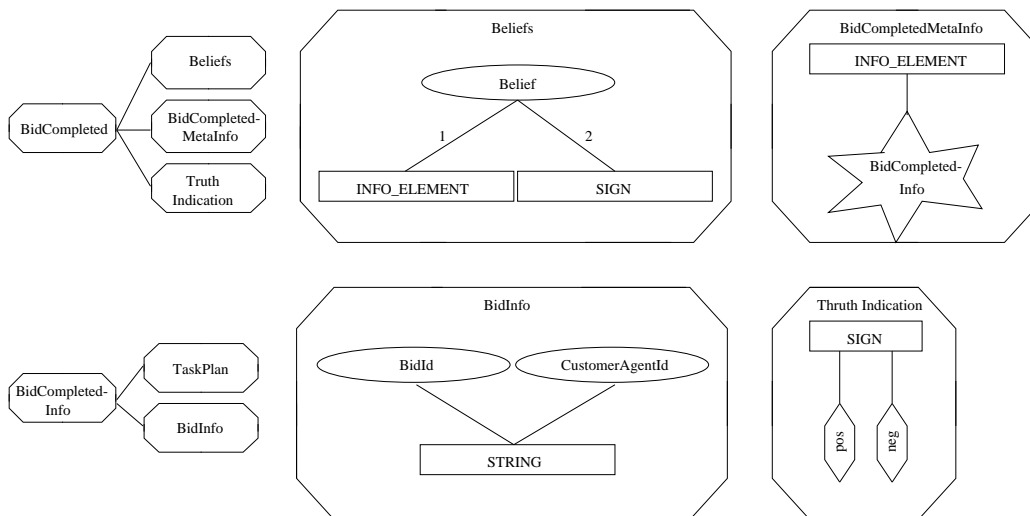


Figure 24: Information type: BidCompleted

TimeWindow The information type is shown in figure 25. As can be seen, there are five relations. The first relation is *TaskId*, this relation specifies the identity of the task for which the time window holds. It uses the sort STRING to represent the identity of the task. The other four relations have already been discussed in the *Bid* information type.

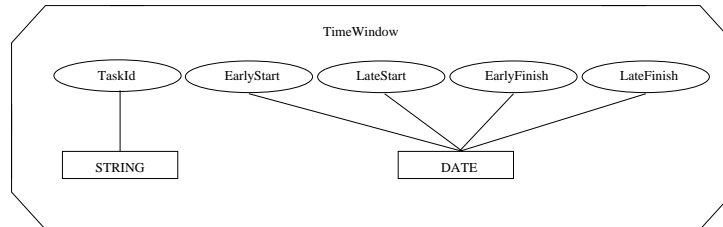


Figure 25: Information type: TimeWindow

Role The *Role* information type is shown in figure 26 and consists of one relation: *ComponentRole*. This relation denotes the role the component has within the SUPPLIERAGENT and uses the sort STRING. This sort is used to represent the role.

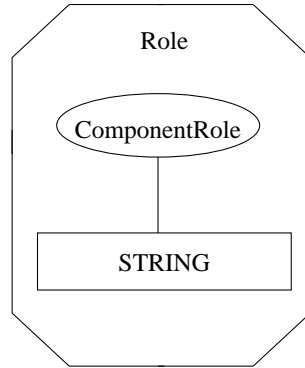


Figure 26: Information type: Role

5.4 Design using UML

Now that we have made a design on a higher level, we are ready to treat the detailed design. For this UML is used within this document. In the first section the Class Diagram of the SUPPLIERAGENT is presented. The mapping between the higher level design discussed previously and the more detailed design as presented in this section is done in the following way: The components have become classes and the flow of information from one component to another component is done through the parameters of a method call. When the flow of information is synchronous, the method call returns the result. When the communication is asynchronous, the information is put through a method call in the requesting component. A combination of both is also possible, for example when you subscribe to a particular event you will get a confirmation through a synchronous return value. However, the events the subscription was about are put in an asynchronous way. The information types as defined in the previous section have been modeled as classes and the information included in the information types can be accessed through method calls.

5.4.1 Class Diagram

In figure 27 the Class Diagram of the SUPPLIERAGENT is shown. The SUPPLIERAGENT contains the COMPONENTMANAGER. Each SUPPLIERAGENT contains precisely one COMPONENTMANAGER and each COMPONENTMANAGER is contained within precisely one SUPPLIERAGENT. This is done be-

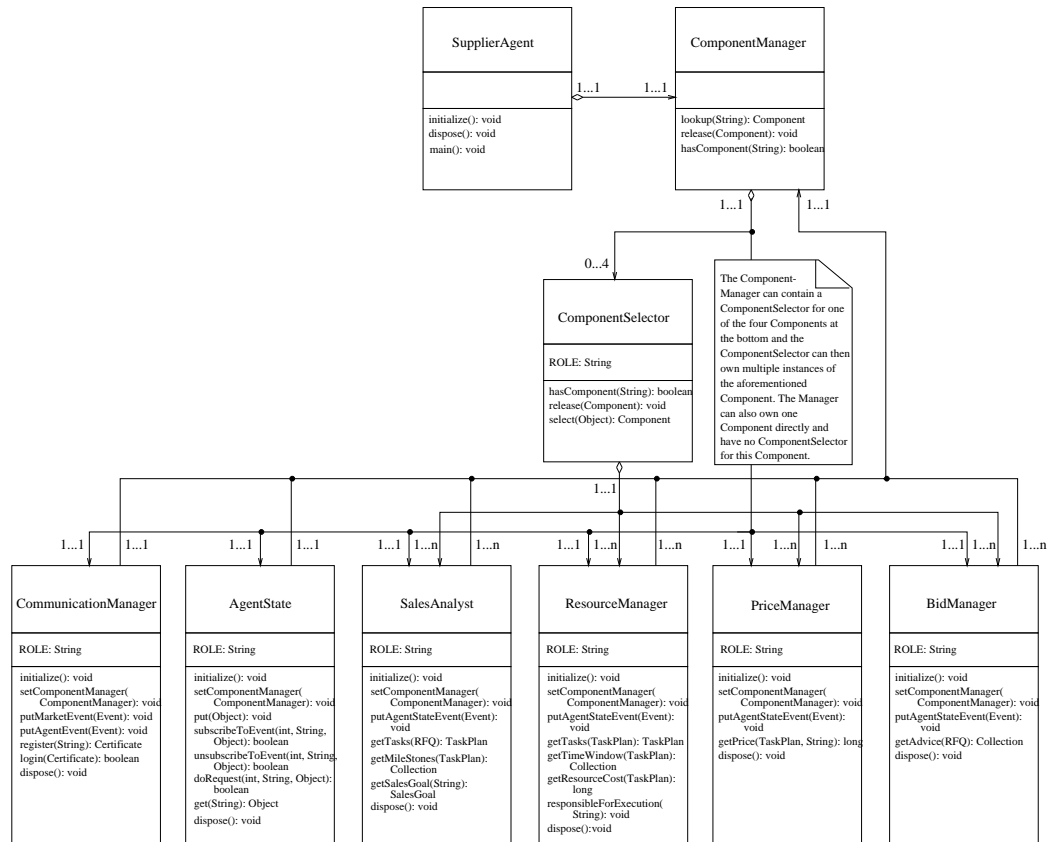


Figure 27: The UML Class Diagram for the Supplier Agent

cause all the components the SUPPLIERAGENT needs are managed by one COMPONENTMANAGER. And we've modelled the COMPONENTMANAGER to manage only those components for one SUPPLIERAGENT. The COMPONENTMANAGER can have a COMPONENTSELECTOR for every component it is supposed to manage (except for the COMMUNICATIONMANAGER and the AGENTSTATE components), but this is not obligatory. If there is only one instance of the component, there is no need to use a COMPONENTSELECTOR for that component. Since there are four components which can have multiple instances, the COMPONENTMANAGER can have at most four COMPONENTSELECTORS. The COMPONENTSELECTOR is contained within precisely one COMPONENTMANAGER because it only needs to keep track of the instances of the components this COMPONENTMANAGER manages. The COMPONENTSELECTOR contains one or more instances of one of the components at the bottom (SALESANALYST, RESOURCEMANAGER, BIDMANAGER and PRICEMANAGER). And these components each have precisely one COMPONENTSELECTOR. If the COMPONENTSELECTOR is not used for a component, this component is contained within precisely one COMPONENTMANAGER, and the COMPONENTMANAGER contains precisely one instance of this component. All components have a reference to the COMPONENTMANAGER they're managed by (which can be only one). There can be multiple instances of the PRICEMANAGER, RESOURCEMANAGER, SALESANALYST and BIDMANAGER and each of these instances has a reference to the COMPONENTMANAGER. There is only one instance of the COMMUNICATIONMANAGER (and therefore it can have only one reference to the COMPONENTMANAGER) because the Server should have only one instance of a component to communicate with, so there can be only one COMMUNICATIONMANAGER. However if there can be multiple markets, it would be useful to have an instance of the COMMUNICATIONMANAGER for every market, this however is not modelled within this paper. There is also one instance of the AGENTSTATE because the events should be handled by only one component.

The interfaces of all of the classes are also shown in the figure.

5.4.2 Sequence Diagram

Sequence Diagram of the UseCase Compose Bid In figure 28 the Sequence Diagram of use case *Compose Bid* for the SUPPLIERAGENT is shown.

A prerequisite is that an RFQ has already arrived at the COMMUNI-

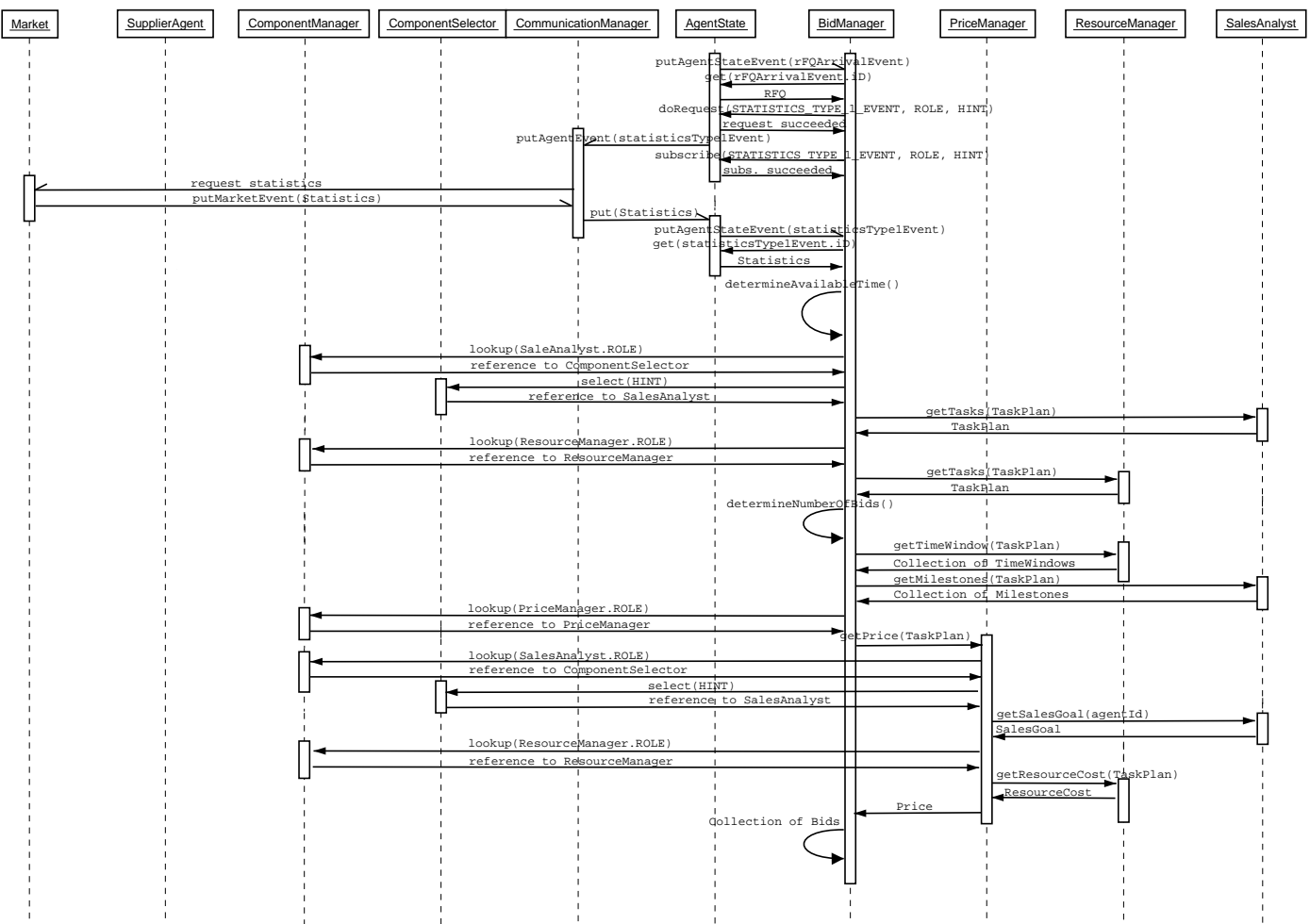


Figure 28. The UML Sequence Diagram for the Supplier Agent: The *Component Bids* UseCase

CATIONMANAGER. The assumption has been made that the BIDMANAGER has already got a reference to the COMMUNICATIONMANAGER and the AGENTSTATE. First of all the BIDMANAGER retrieves the RFQ from the AGENTSTATE and start with the making of the bid. For this, it may want to use statistics from the market. If so, the BIDMANAGER requests these statistics from the AGENTSTATE. After the request has been done, the BIDMANAGER subscribes to the event for the arrival of these particular statistics. This component checks if the statistics are available within the AGENTSTATE (if this is the case, an event is passed right away by the AGENT STATE), or have to be retrieved from the MARKET. If the statistics have to be retrieved from the MARKET, the AGENTSTATE passes an event to the COMMUNICATIONMANAGER. The COMMUNICATIONMANAGER interprets the event and retrieves the statistics from the MARKET. After the statistics have been received, the COMMUNICATIONMANAGER passes these through to the AGENTSTATE. This component notifies all subscribed components of the arrival of the statistics, including the BIDMANAGER. Now that the BIDMANAGER knows that the statistics are present it retrieves them from the AGENTSTATE. After this has been done it determines the available time for the bidding process. The assumption has been made that the RESOURCEMANAGER knows the current market time. Subsequently, the SALESANALYST is used to give the BIDMANAGER an advice concerning which tasks to focus on when taking the customer into account. For this the BIDMANAGER has to retrieve the reference to the SALESANALYST from the COMPONENTMANAGER and, when multiple instances exist, the COMPONENTSELECTOR is also used. After the reference has been received the BIDMANAGER passes the RFQ containing tasks to be considered to the SALESANALYST. The SALESANALYST executes the aforementioned task and return the resulting TaskPlan. The reference to the RESOURCEMANAGER is retrieved in the same way as with the SALESANALYST. The BIDMANAGER passes the TaskPlan through to the RESOURCEMANAGER. After this has been done, the RESOURCEMANAGER selects the tasks which can be bid on when looking at the resources. The resulting TaskPlan is retrieved and is used to determine the number of bids. Subsequently the composition of these bids is determined and after that the time windows included in the bids are fixed. In order to be able to do this, the RESOURCEMANAGER is consulted, since it knows the time on which the resources are available. Another issue is what milestones to include in the bids which is done next. For this, the SALESANALYST is asked for advice, because the SALESANALYST knows the preferences of

the customer. Now that the bids have almost been constructed the price to be included needs to be calculated. For this the PRICEMANAGER is asked for advice. The reference is retrieved in the same way as described above and the TaskPlan and the agentId are passed to the PRICEMANAGER. The PRICEMANAGER asks for the SalesGoals for this customer from the SALE-ANALYST and the RESOURCEMANAGER is consulted for the resource costs of the resources included in the TaskPlan. The references are retrieved in the same way as with the BIDMANAGER. After the PRICEMANAGER has received the advice it determines the price and return it. Now the BIDMANAGER is able to make the Collection of Bids to be done and stores them until the time has come to issue them.

Sequence Diagram of the UseCase Bid Award In figure 29 the Sequence Diagram of use case *Bid Award* for the SUPPLIERAGENT is shown.

A precondition is that a bid has been issued. First the COMMUNICATIONMANAGER is notified that it's bid has been awarded by passing the bidAwardEvent. Now the COMMUNICATIONMANAGER updates the AGENTSTATE with this Event. The assumption has been made that the RESOURCEMANAGER is subscribed to this particular event. The RESOURCEMANAGER is now notified by the AGENTSTATE that the bid that has been issued is awarded. After the deposit has been paid, the RESOURCEMANAGER reserves the necessary resources. When the AGENTSTATE notifies the RESOURCEMANAGER that the earliest time to start has arrived, the RESOURCEMANAGER starts with the execution of the tasks that were awarded (as can be seen in the UseCase *Execute Task*). Whenever a milestone is reached, this is communicated to the MARKET by putting the milestone into the AGENTSTATE component. The AGENTSTATE in turn passes an event through to the COMMUNICATIONMANAGER. The COMMUNICATIONMANAGER interprets the event and communicates the milestone to the MARKET. After that, the RESOURCEMANAGER continues with the completion of the rest of the tasks. After the execution of the tasks is completed, the RESOURCEMANAGER releases the resources it reserved for the tasks. Now the RESOURCEMANAGER communicates a TaskPlan which contains the tasks that have successfully been executed to the MARKET (the communication is done in the same way as described above). When the latest time to finish arrives, the AGENTSTATE notifies the RESOURCEMANAGER of it. The RESOURCEMANAGER waits for the payment to be done. When the payment has been done, the RESOURCE-

MANAGER marks the bid as *done*.

6 Implementation

After the design that has been presented the implementation phase is next. In this section a brief overview of the problems and choices that have been made during this phase is given. First, the issues involved in implementing each of the information types and components are treated.

6.1 Implementing the components

In this section the issues involved in implementing the information types and each of the components are passed. First of all, the information types are discussed and after that the components. Since we focus on four components we will not go into further detail for the components we didn't implement. The components that we have implemented are: the `COMMUNICATIONMANAGER`, the `AGENTSTATE`, the `BIDMANAGER` and the `SUPPLIERAGENT`.

6.1.1 Implementing the information types

The implementation of the information types was the first step in the implementation phase. We've chosen to implement these first because all components use the information types and the information types can be implemented without needing any other classes. Within *Java* classes are placed within packages, we've chosen to split the information types into two packages. The information types concerning events are put in the `edu.umn.magnet.client.supplier.event` package and the rest of the information types have been put in the `edu.umn.magnet.client.supplier.infotypes` package. When an information type is not included in this section but was part of the DESIRE section on information types it was already part of the MAGNET system or part of the Java programming language.

Within the `edu.umn.magnet.client.supplier.event` the following information types have been implemented:

- BidEvent
- MarketEvent
- PaymentEvent
- RFQEvent

- `StatisticsEvent`
- `SupplierAgentEvent`

There are two basic types of events: **SupplierAgentEvent** and **MarketEvent**. Both of these were not part of the model as presented in the design. The **SupplierAgentEvent** is used as a superclass for the events that can occur within the `SUPPLIERAGENT`. The class has been created because the components should be able to receive all possible events within the `SUPPLIERAGENT` and with a superclass as the aforementioned only one method is needed to receive all these events. The **MarketEvent** has been added because we need to be able to communicate with the Market and these are not part of the internal events of the `SUPPLIERAGENT`. The **MarketEvent** can include all elements that need to be communicated to and from the `SUPPLIERAGENT` like RFQ's and bids. The **SupplierAgentEvents** have been split up into four events since they use the same attributes. The **BidEvent** is used to pass events concerning bids and has a number of event types. The **PaymentEvent** is used to pass events concerning payments for the execution of tasks within bids. Third, the **RFQEvent** is used to model events concerning RFQ's. Finally, the **StatisticsEvent** models events about statistics. The implementation of the aforementioned classes didn't cause any difficulties.

Within the `edu.umn.magnet.client.supplier.infotypes` the following information type are included:

- `Certificate`
- `ComponentId`
- `MileStone`
- `SalesGoal`
- `TimeWindow`

The classes correspond to the information types as mentioned in the design. The **Certificate** information type was not included in the design and is an information type we are not responsible for. The **ComponentId** information type is used for database purposes and is simply a class that groups the **ROLE** of a component and it's **HINT**. The implementation of the classes didn't cause any problems.

6.1.2 Working with Threads

After the implementation of the information types, the components are next. Since the SUPPLIERAGENT should be able to handle multiple RFQ's at the same time, for which multiple BID MANAGERS can be used, we need to have some kind of method for components to run at the same time. Within Java you can use Threads to achieve this goal. Using this scheme results in each agent running separately and when necessary they can run at the same time. Since we were not familiar with Threads a brief explanation is given here.

A Thread in Java is a single sequential flow within a program. In Java, the program usually executes within one Thread. When you want to have multiple Threads you can extend the Thread class and by doing this add an extra Thread to the execution. In figure 30 the lifecycle of the Thread is shown. The Thread is created when the run method is invoked in the class that is extending the Thread class, this is done by calling start(). The Thread is terminated when the method has fully executed the code within the run method. Threads have a priority, the amount of CPU they each get depends on their priority. When you want to state that other Threads are allowed to run instead of your Thread, you can use the yield method. Threads can be created dynamically, so in our system when you start up a BIDMANAGER you can simply create a new Thread and let the BIDMANAGER run in this Thread.

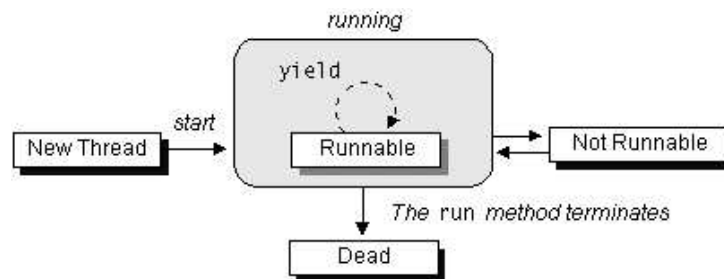


Figure 30: The Lifecycle of a Thread

When reading the documentation about Threads in Java we noticed that it is preferred to have the code concerning events in one Thread. If, as a consequence of the event, a rather large piece of code has to be executed, a separate Thread has to be started. This Thread can be started in the method invoked by the event. For information on Threads see [Sun Microsystems inc.,

1995].

6.1.3 Implementation of the AgentState component

The AGENTSTATE component is the most difficult component to implement so we decided on doing this component first. The main focus of the research on the SUPPLIERAGENT within the MAGNET project is going to be on the bidding. Since this is the case, the implementation doesn't need to support the events taking place after the BidAward. Another thing we didn't implement are the issues related to Statistics, since Statistics are not supported in the MARKET yet. Now the issues that arose when implementing the AGENTSTATE are described.

First of all, we started implementing the subscription to the events. For this, we had to create a database to keep track of all the subscriptions. The database needs to be able to keep track of subscriptions and give all the components that need to be notified when a particular event occurs. We've implemented the database in a very simple way, just by using Vectors to maintain the subscriptions. Since the subscriptions include the type of event, the role of the component and its hint this information has to be grouped so it can be put in a Vector. We've chosen to make a class named **ComponentId** which contains the role and the hint of the component that has subscribed. To group the eventtype and the identification together we introduced another class, namely **SubscriptionNode**. We've chosen to construct the **ComponentId** class because with this it is easier to return a Vector with the subscribed components, since an entry in the Vector can contain only one Object. If we had just included the role, hint and type of event in the **SubscriptionNode** class this wouldn't have been as easy.

Secondly, we started implementing the *put()* and *get()* methods. When we started implementing these, it became clear we needed a database to store the information that was put and retrieved. We've chosen to use a temporary solution, since there is someone who is working on databases within the MAGNET project. The temporary solution was to just store the bids, bidwards and RFQ's in a Vector. Another issue that arose was that for the putting and retrieving method an id was used and in case of multiple markets the id is not guaranteed to be unique. Since the research doesn't focus on multiple markets for now, we've assumed that the id was unique. If multiple markets need to be supported, the identity of the protocol element and the identity of the market can be used as a composite key. When the

put method is called, the appropriate actions are taken such as saving the Object that was put and: In case of a bid, the COMMUNICATIONMANAGER is notified it should be sent. In case of an RFQ a BIDMANAGER is assigned to determine the bids for this particular RFQ. When the *get()* method is called, the appropriate protocol element is looked up in the Vectors and returned.

After this had been implemented, the next thing we've implemented was the selection of the BIDMANAGER when a new RFQ arrives. In order to know what BIDMANAGERS are available we added an extra method to the interface, since it's impossible to determine this just by using the COMPONENTMANAGER. When this method was implemented, we decided to add an extra class for keeping track of the available BIDMANAGERS. With this class, you can state the available BIDMANAGERS and request an available one when a new RFQ arrives. When this is done, the BIDMANAGER that is returned is assigned to the RFQ. A BIDMANAGER can be made available again by passing the id of the RFQ it has been assigned to. The algorithm for selecting an available BIDMANAGER is very simple, just take the first one that is available.

Now that the selection of the BIDMANAGERS have been arranged we need a mechanism to make a BIDMANAGER available again. The appropriate time to make the BIDMANAGERS available again is when the RFQ bid deadline arrives. This is because no bidding is allowed after this deadline, so there's no need for the BIDMANAGER to issue bids any more. Within the MAGNET system there is a class that can be used to issue events when a particular time passes called TimeMap. The problem that arose when we used the class was that the events that were thrown didn't contain any information. Since we want to include multiple time events concerning an RFQ and a bid we need some method to be able to pass information on what time event has arrived. We first came up with using a regular sequence in which the time events occurred. With this we kept track of how many events had occurred and just sent the next one in the sequence to the AGENTSTATE. We didn't like the assumption that the sequence would always be the same, so we looked at the code that had been written in the MAGNET system and used the TimeMap. We used this code and now the assumption is dropped, since with this code the events can contain an eventtype.

After the previous implementation issues, we noticed that the AGENTSTATE issues a lot of events. Since we want to be able to handle a lot of events at the same time, it wouldn't be usefull for components to receive all events of

a particular type since they are interrupted when an event occurs. That's why we introduced the *expression of interest*, with this components have to express interest in a particular id of a protocol element. Components only receive those events they have subscribed to and that event should also be concerned with an id they have expressed interest in. If components do not express interest in an id, they are not able to receive events. There might be some components that are interested in all events of a certain type, for example, if there is only one RESOURCEMANAGER it needs to receive all bidaward events, since it manages all resources. That's why we've introduced the INTERESTED_IN_ALL constant, with which a component can express interest in all protocol elements. Finally, we introduced a constant REMOVE_ALL to easily remove all expressions of interest.

6.1.4 Implementation of the BidManager component

Since the AGENTSTATE assigns a BIDMANAGER to a newly arrived RFQ, the next component to be implemented was the BIDMANAGER. Since the SUPPLIERAGENT should be able to handle multiple RFQ's, we've decided to make an extra class, contained in the BIDMANAGER, namely **BidDeterminer** which actually determines the bids. This class extends the Thread class so when the BIDMANAGER is started up a separate Thread is started up, in which the collection of bids is determined. Events for the BIDMANAGER are received in the same Thread where the AGENTSTATE component runs. Since the determination of the bid is done in a different Thread, we need to have a way of communication between the Threads. The normal way to do this, is by using a synchronized queue between the Threads. We've implemented our queue as a FIFO-queue, since we want the **BidDeterminer** to handle the events in the order they were thrown.

The **BidDeterminer** we've implemented is only useful for testing purposes and therefore it only constructs one bid in which all the advices are included. For this we've implemented the following functionalities: retrieving the RFQ from the AGENTSTATE, asking for advice concerning the tasks to bid on from the SALESANALYST and the RESOURCEMANAGER, getting the timewindows from the RESOURCEMANAGER, getting the milestones from the SALESANALYST, getting the price to bid from the PRICEMANAGER and submitting the bid. The **BidDeterminer** also needs to be able to subscribe to events, this means the AGENTSTATE must be accessible from different Threads. This, however, isn't a problem because we use the Avalon sys-

tem. It guaranties that no component can be accessed by more than one component at the same time and because of this, components in different Threads are synchronized. The only difficulty we experienced during the implementation was with the Milestones, this was because we didn't know how to include them in a bid. At first we thought that a Milestone could be added to a bid by adding a taskId with a MilestoneId. However, this wasn't the proper way to do this. A milestone is just a taskID for which either the duration of the execution of the task is specified or the specific start and finish times. Another task can be included in the milestone by specifying the id of this task and the id of the milestone task.

6.1.5 ResourceManager, SalesAnalyst and PriceManager

In order to be able to test the SUPPLIERAGENT we need these components to be implemented. Since other people are responsible for implementing these components, we've decided to implement dummy components. The dummy components don't behave intelligently but just return some random information in the correct format. Because of this the implementation was straightforward and didn't cause any problems.

6.1.6 CommunicationManager

Now that all the components have been implemented the SUPPLIERAGENT needs to be able to communicate, therefore we now implemented the COMMUNICATIONMANAGER. Since the person responsible for the security of the SUPPLIERAGENT wasn't finished with his part, we didn't implement the register and login functions. Since these functions aren't implemented, we've added the setAgentId function. This was necessary to be able to identify a SUPPLIERAGENT when testing with multiple SUPPLIERAGENTS. It is capable of communicating with outside entities. For testing purposes we provide the COMMUNICATIONMANAGER with the reference to the outside entity. Communication goes by passing MarketEvents, these events contain the object that is to be communicated as it's contents. To be able to receive MarketEvents the COMMUNICATIONMANAGER needs to extend the MarketEventListener interface. If there are multiple SUPPLIERAGENTS, the passing of the MarketEvents between the SUPPLIERAGENTS and the outside entity needs to be synchronized. In order for the COMMUNICATIONMANAGER to receive SupplierAgentEvents it also implements the AgentStateEventLis-

tener interface. The implementation of this component didn't cause any problems.

6.1.7 SupplierAgent

The next step was to have a component that creates all the components and puts them in a manager. With the Avalon system a XML file can be used to create all the instances of the components. The XML file looks like this:

```
<MAGNET>
<component
role="edu.umn.magnet.client.supplier.component.BidManagerSelector"
class="org.apache.avalon.excalibur.component.ExcaliburComponentSelector">
  <component-instance name="bidmanager1"
    class="edu.umn.magnet.client.supplier.component.DefaultBidManager">
  </component-instance>
  <component-instance name="bidmanager2"
    class="edu.umn.magnet.client.supplier.component.BidManager2">
  </component-instance>
</component>
<component
role="edu.umn.magnet.client.supplier.component.SalesAnalyst"
class="edu.umn.magnet.client.supplier.component.DummySalesAnalyst">
</component>
<component
role="edu.umn.magnet.client.supplier.component.ResourceManager"
class="edu.umn.magnet.client.supplier.component.DummyResourceManager">
</component>
<component
role="edu.umn.magnet.client.supplier.component.AgentState"
class="edu.umn.magnet.client.supplier.component.DefaultAgentState">
</component>
<component
role="edu.umn.magnet.client.supplier.component.CommunicationManager"
class="edu.umn.magnet.client.supplier.component.DefaultCommunicationManager">
</component>
<component
role="edu.umn.magnet.client.supplier.component.PriceManager"
class="edu.umn.magnet.client.priceManager.SAPriceManager">
</component>
</MAGNET>
```

With this file, two instances of the BIDMANAGER are created. They each use a different class file and are put in the same EXCALIBURCOMPONENTSELECTOR. All the other components have one instance and therefore don't use the EXCALIBURCOMPONENTSELECTOR. The file is used as input for the ConfigurationBuilder.

After the configuration has been built, the ExcaliburComponentManager is configured using the aforementioned configuration. We have to use this

manager instead of the `DefaultManager` because that one is not configurable with a `ConfigurationBuilder`. After the manager has been configured, the components are provided with the necessary information to be able to work properly. This includes putting the reference to the `ComponentManager` and setting the hints of the components. After this, all components are initialized. Implementing this part of the system caused some problems, in particular the configuration with the XML file. The classes included in the XML file have to be part of the `edu.umn.magnet.client.supplier.component` package because else they aren't loaded. This has something to do with the `ClassLoaders` that are used for different parts of the program. Another problem that came up during the implementation was that every time a component was used a new instance of the component was created. This was not something we wanted to have, since the `PRICEMANAGER` for example is going to use an algorithm based on history. Since the `PRICEMANAGER` needs to keep track of this history for itself, the history is lost when a new instance is created. To solve this problem, we checked the Avalon documentation and noticed that when a class implements the `ThreadSafe` interface, no new instance is created. This is exactly what we want, so we implemented the interface in all components. Ofcourse, when using this interface, the components have to be `ThreadSafe`. Since the `ComponentManager` only provides a reference to the components when they are available (which means not being used by another component), this is not a problem.

Now that an overview has been given of the implementation issues, we are going to give a brief overview of how the `Threads` are organized. This is shown in figure 31.

The `PRICEMANAGER`, the `RESOURCEMANAGER` and the `SALESANALYST` aren't shown in the picture, because they can be in any `Thread` in the `SUPPLIERAGENT`. When one of these components is referenced from a particular `Thread`, the component flows into this `Thread`.

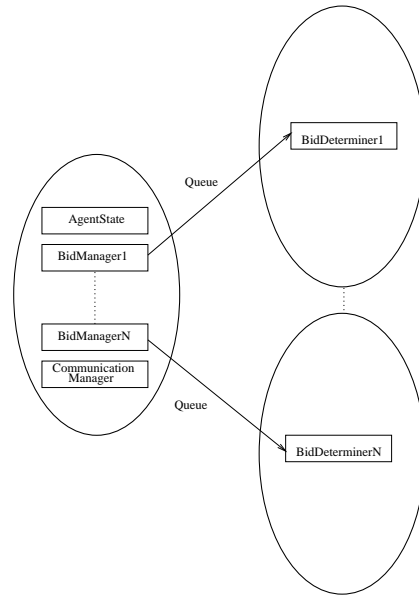


Figure 31: The Threads within the Supplier Agent

7 Testing

Now that we've implemented all the components, we are ready to test the implementation. We haven't tested all the classes, because some of them are trivial to implement, for example the RFQEvent is real easy to implement. In every section a testplan is presented which has been used to test the class, after which the result is shown.

7.1 BidManager

Within the BIDMANAGER component the following classes have been tested:

- **Queue** This is the queue used to pass events between the BidManager Thread and the Algo Thread.

7.1.1 Queue

First of all, the eventqueue has been tested.

Basic putting and getting of items

- **Name:** Putting an object in the queue and retrieving it.
- **Goal:** Determine whether the putting of an object works properly and it can be retrieved correctly.
- **Input:** Set of Strings containing a certain word.
- **Predicted output:** The Strings contain the same word and are retrieved in the same order they were put.
- **Execution conditions:**
- **Approach:** The Strings will be put and retrieved in random order.

Evaluation of the test case

```
q.putObject("object1");
System.out.println((String) q.getHead());
q.putObject("object2");
q.putObject("object3");
System.out.println((String) q.getHead());
```

```
q.putObject("object4");  
System.out.println((String) q.getHead());  
System.out.println((String) q.getHead());
```

Output:

```
object1  
object2  
object3  
object4
```

The output is correct.

7.2 AgentState

Within the AGENTSTATE component the following classes have been tested:

- **EventDatabase** This class is used to maintain the subscriptions to the events.
- **BidManagerAllocation** This class is used to maintain the availability of the BIDMANAGERS.

EventDatabase EventDatabase is the class responsible for maintaining the subscriptions and interest of components and instances of components.

- **Name:** Putting subscriptions and expressions of interest and retrieving them from the database.
- **Goal:** Determining whether the putting of subscriptions and expressions of interest works properly and the retrieving works correct.
- **Input:** Random subscriptions to events and expressions of interest.
- **Predicted output:** The subscribed and interested components are properly stored and returned.
- **Execution conditions:**
- **Approach:** The subscriptions and expressions are put and retrieved.

Evaluation of the test case

```

e.addProtocolElementId("RFQ1");
e.addProtocolElementId("RFQ2");
e.putComponent("Role1", "Hint1", "RFQ1");
e.putComponent("Role1", "Hint2", "RFQ1");
e.putComponent("Role2", "Hint1", "RFQ2");
e.putComponent("Role3", "Hint1", AgentState.INTERESTED_IN_ALL);
e.putSubscription("Role1", "Hint1", 1);
e.putSubscription("Role2", "Hint1", 2);
e.putSubscription("Role3", "Hint1", 2);
e.putSubscription("Role3", "Hint1", 1);
e.putSubscription("Role1", "Hint2", 1);
Vector listeners = e.getEventListeners(2, null);
printVector(listeners);
listeners = e.getEventListeners(2, "RFQ2");
printVector(listeners);
listeners = e.getEventListeners(1, "RFQ1");
printVector(listeners);
e.removeSubscription("Role1", "Hint1", 1);
listeners = e.getEventListeners(1, "RFQ1");
printVector(listeners);
e.removeSubscription("Role3", "Hint1", 2);
listeners = e.getEventListeners(2, "RFQ1");
printVector(listeners);
e.removeComponent("Role3", "Hint1", AgentState.REMOVE_ALL);
listeners = e.getEventListeners(1, "RFQ1");
printVector(listeners);

```

Output:

```

-----
The role of the component: Role2. The hint of the component: Hint1
The role of the component: Role3. The hint of the component: Hint1
-----
The role of the component: Role2. The hint of the component: Hint1
The role of the component: Role3. The hint of the component: Hint1
-----
The role of the component: Role1. The hint of the component: Hint1
The role of the component: Role1. The hint of the component: Hint2
The role of the component: Role3. The hint of the component: Hint1
-----
The role of the component: Role1. The hint of the component: Hint2
The role of the component: Role3. The hint of the component: Hint1
-----
The role of the component: Role1. The hint of the component: Hint2

```

The output is correct.

BidManagerAllocation The BidManagerAllocation is used to maintain the availability of BidManagers.

- **Name:** Allocation of bidmanagers for RFQ's.

- **Goal:** Determine whether the putting of new BidManagers, the allocation for a new RFQ and the releasing of a BidManager work properly.
- **Input:** Random hints for BidManagers and RFQ's.
- **Predicted output:** The BidManagers are properly allocated and released.
- **Execution conditions:**
- **Approach:** The BidManagers are put, allocated and released.

Evaluation of the test case

```
Vector v = new Vector();
v.add("Hint1");
v.add("Hint2");
v.add("Hint3");
v.add("Hint4");
b.setNewBidManagers(v);
String s = (String) b.getAvailableBidManager("RFQ1");
System.out.println("String now " + s);
s = (String) b.getAvailableBidManager("RFQ2");
System.out.println("String now " + s);
b.removeAllocationOfBidManager("RFQ2");
s = (String) b.getAvailableBidManager("RFQ3");
System.out.println("String now " + s);
b.removeAllocationOfBidManager("RFQ1");
s = (String) b.getAvailableBidManager("RFQ4");
System.out.println("String now " + s);
s = (String) b.getAvailableBidManager("RFQ5");
System.out.println("String now " + s);
s = (String) b.getAvailableBidManager("RFQ6");
System.out.println("String now " + s);
s = (String) b.getAvailableBidManager("RFQ6");
System.out.println("String now " + s);
```

Output:

```
String now Hint1
String now Hint2
String now Hint2
String now Hint1
String now Hint3
String now Hint4
String now null
```

Output is correct.

7.3 The complete system

The complete system has been tested in this section.

The test for the complete system

- **Name:** Determining a bid for a particular RFQ.
- **Goal:** Check to see if the SupplierAgent can determine a bid as response to a certain RFQ.
- **Input:** Random RFQ's from a random class.
- **Predicted output:** Two bids are determined.
- **Execution conditions:**
- **Approach:** The RFQ's are communicated and the resulting bids are printed.

Evaluation of the test case

```

TaskPlan p = new TaskPlan();

try{
    p.addTask(s);
    p.addTask(t);
}catch(InvalidTaskException e){
    System.out.println("RUNEXPERIMENT: Error while adding tasks");
}

p.setPlanName("Test");
RFQ rFQ = new RFQ();
RFQ rFQ2 = new RFQ();
ProtoElementIdGenerator generator = ProtoElementIdGenerator.getInstance();
rFQ.setId(generator.getUniqueId());
rFQ.setBidDeadline(d1);
rFQ.setEarliestConsider(d2);
rFQ.setEarliestOffer(d3);
TaskPlan tp = new TaskPlan(p);
rFQ.setTaskPlan(tp);
// Do the same for RFQ2, not shown here.

SupplierAgent supplierAgent = new SupplierAgent();
supplierAgent.initialize();
supplierAgent.setAgentId("Someone");

communicationManager.putMarketEvent(new MarketEvent(this, rFQ));
communicationManager.putMarketEvent(new MarketEvent(this, rFQ2));

```

Output:

```
Bid is sent to the market with id: bidmanager12 and RFQ id: 1 and
  SupplierAgentId: Someone
Bid is sent to the market with id: bidmanager13 and RFQ id: 0 and
  SupplierAgentId: Someone
```

Output is correct.

7.4 Conclusion

More testing has been done during the running of the experiments, but only the most significant cases have been shown. All these test cases have succeeded and therefor the *supplier agent* seems to be working fine.

8 The SalesAnalyst

In this section we are going into more detail on the SALESANALYST component. We describe the construction of a SALESANALYST that learns from the developments on the Market and previously submitted bids. Another issue involved in the SALESANALYST is what the possible salesgoals are, and what they stand for.

8.1 Analyzing the Sales Analyst

8.1.1 Proposals for learning schema's

When thinking of learning schema's within the SALESANALYST, we came up with the idea to use datamining techniques to analyse information for the SALESANALYST. The information which can be used is maintained within the SALESANALYST since we use private information such as information on payments.

The above described information can be used for several things:

- The datamining techniques can be used to derive the profiles of the customers. For example, you could derive the profile of good and bad customers. A customer is called a good customer if he pays in time and a bad customer otherwise. When deriving a profile for each of these types of customers, you can look at their behaviour considering what times are included in the RFQ's they issued and the tasks they usually want to have executed.
- The derived profiles can be used to advice the BIDMANAGER not to bid on certain tasks, because the risk is too high. For example, when the current customer is a bad customer and bad customers usually don't pay for this task, it could advise not to bid on this task.
- Another piece of information that can be derived from the profile is what salesgoal to communicate to the PRICEMANAGER. For example, when the SALESANALYST has determined it is wishful to get this customer as a regular customer, it could advice to bid a lower price. When doing this, the customer could be binded because he likes the lower price.

- The milestones can be determined using the profile. What milestones to include can be derived using the bids and the awards of the bids, combined with the profile of the customer. Using the datamining schema, you can derive rules about what milestones are preferred by this type of customer.
- When using the datamining technique, you can check for regularities in the behaviour of the current customer, if known. For example, a particular customer only awards bids which cover a few tasks. It can be derived that it is better to issue bids which cover only a few tasks.
- Another feature could be that a supplier can have a strategy to conquer a certain position within a market. To get such a position, he might want to use an aggressive strategy, and examine the information available on how to apply this in the best way.
- Finally, the SALESANALYST can advise the RESOURCEMANAGER on how to allocate resources for this particular bid. For example, when the customer is considered not very reliable, you could pass the advice not to allocate the resources for the full 100%.

8.1.2 Salesgoals

Now that we've identified schema's with which the SALESANALYST can learn, we need to identify the salesgoals that can follow from the aforementioned learning schema's. We came up with the following salesgoals:

- eager to sell
- bind customer
- maintain as regular customer
- reduce risk
- not eager to sell
- none

eager to sell This salesgoal is used to model that the SALESANALYST wants these tasks to be sold.

bind customer When the SALESANALYST wants to bind a customer and make the customer a regular one, it can pass this salesgoal to the PRICEMANAGER.

maintain as regular customer When the customer is already a regular customer the SALESANALYST uses this salesgoal to notify the PRICEMANAGER on maintaining this relation.

reduce risk This salesgoal is used to inform the PRICEMANAGER that this customer is bad, and it should adjust the price accordingly so that the risk is reduced.

not eager to sell When the product is rare or the SUPPLIERAGENT doesn't have much of this product in stock, it can pass this salesgoal to the PRICEMANAGER.

none When the SALESANALYST decides that there is no particular salesgoal, it can pass this one.

8.1.3 Some definitions

Now we need to define some concepts, we've already introduced these concepts, but they need to be defined more specific.

good and bad customer First of all, we define the concept of a bad customer. *A customer is a bad customer if the customer didn't pay in time twice or more.* If a customer is not a bad customer, it is defined as a good customer. We've chosen this definition for a bad customer because one time could be a coincidence, and two times makes it more structural. To set the number of bad payments to more than two would be too much, because else the SUPPLIERAGENT would have to suffer with too many bad payments.

regular customer Now the concept of a regular customer is defined. *A regular customer is a customer that awards one or more bids per RFQ in more than 50 % of the cases.* We've chosen more than 50 % because the customer prefers to assign a task to you for most of the RFQ's it issues.

8.1.4 Necessary data

First of all, we've determined what data we need, and where we expect the data to come from.

Within the `SUPPLIERAGENT` we expect to have the following information available:

- All RFQ's that have been received by the `SUPPLIERAGENT`.
- Information on bids issued for a particular RFQ.
- Whether the bid has been awarded or not.
- Information on the execution of the tasks included in a bid which has been awarded.
- Information on the payment of the awarded bids.
- History of the customers with this `SUPPLIERAGENT`.

Other information the `SALESANALYST` needs is assumed to be provided by the `MARKET`:

- General information on tasks, this includes information on how often this task is requested etc.

8.1.5 Communication with other components

The way the `SALESANALYST` works is as follows: First of all the `SALESANALYST` subscribes to all events concerning RFQ's, bids, bid awards, payments and information on the execution of a task. When an event occurs, it retrieves the information needed from the `AGENTSTATE` and store it in a way suitable for the algorithm. The `SALESANALYST` determines an advice when it is called by the `BIDMANAGER`. In order to be able to give this advice, the `BIDMANAGER` passes the RFQ it wants advice about. Now, the `SALESANALYST` retrieves the information as described in the previous section from the `MARKET`. After that, it asks the `RESOURCEMANAGER` advice on what the load is on the tasks included in the RFQ. Now the `SALESANALYST` determines the advice using it's algorithms. Four advices are determined: First of all, the tasks to be considered are passed to the `BIDMANAGER`. The `RESOURCEMANAGER` receives advice on how to allocate the resources, given the current customer. Third, the milestones to be included are communicated to the `BIDMANAGER`. Finally, the `SALESANALYST` advices the `PRICEMANAGER` on the current salesgoals, given the tasks and the current customer.

8.1.6 Problems with the aforementioned proposals

The problems with the schema described above are that, at the current time, the payments are not supported. In order to derive the profile of good and bad customers we need to know when a customer is good or bad, for which payments are used. Another problem is that the current customer agent doesn't particularly bind to a customer, so there is no need to try and bind a customer. Finally, we would need a lot of data to obtain results, so the SUPPLIERAGENT would have to be on the MARKET for a long time.

8.1.7 Proposed solutions for the problems

The problems that have been described have to be solved so we can have a proper means of testing our implementation. A way to do this, is by using a generator for RFQ's, BidAwards and Payments. These elements don't have to be generated totally random, but could be made suitable for testing our SALESANALYST. We could for example introduce a number of customers, which are slightly different, but fit into the profile of a *good* or *bad* customer. With this, we are able to test the effectiveness of the schema we've introduced.

8.2 Selecting the algorithm

First of all, the datamining algorithm used to derive the aforementioned profiles has been chosen. The dataset to be used is both numerical (the times included in the bid, for example) and categorical (whether a certain task is included in a bid). Since this is the case, we have to use an algorithm that can take both types of input. We've decided to choose the C4.5 algorithm [Quinlan, 1993].

8.2.1 The C4.5 algorithm

The C4.5 algorithm generates a classifier in the form of a *decision tree*. A *decision tree* is a structure that is either:

- a *leaf*, indicating a class, or
- a *decision node* that specifies some test to be carried out on a single attribute value, with one branch and subtree for each possible outcome of the test.

To classify with the *decision tree*, start at the root of the tree and move through it until a leaf is encountered. At a nonleaf node of the tree, the next branch to be taken is determined by the outcome of the test of the node in this particular case. In the next paragraph, a brief explanation on how the tree is constructed is given.

Constructing the *decision tree* The tree, as described above, is constructed with the divide-and-conquer method. The method originates from [Hunt *et al.*, 1966] and deals with constructing a decision tree from a set T of training cases. Let the classes be denoted $\{C_1, C_2, \dots, C_k\}$. There are three possibilities:

- T contains one or more cases, all belonging to a single class C_j :

The decision tree for T is a leaf identifying class C_j .

- T contains no cases:

The decision tree is again a leaf, but the class to be associated with the leaf must be determined from the information other than T . For example, a leaf might be chosen in accordance with some background knowledge of the domain, such as the overall majority class. C4.5 uses the most frequent class of the parent of this node.

- T contains cases that belong to a mixture of classes:

In this situation, the idea is to refine T into subsets of cases that are, or seem to be heading towards, single-class collections of cases. A *test* $T10$ is chosen, based on a single attribute, that has one or more mutually exclusive outcomes $\{O_1, O_2, \dots, O_n\}$. T is partitioned into subsets T_1, T_2, \dots, T_n , where T_i contains all the cases in T that have outcome O_i , of the chosen test. The decision tree for T consists of a decision node identifying the test, and one branch for each possible outcome. The same tree-building machinery is applied recursively to each subset of training cases, so that the i th branch leads to the decision tree constructed from the subset T_i of training cases.

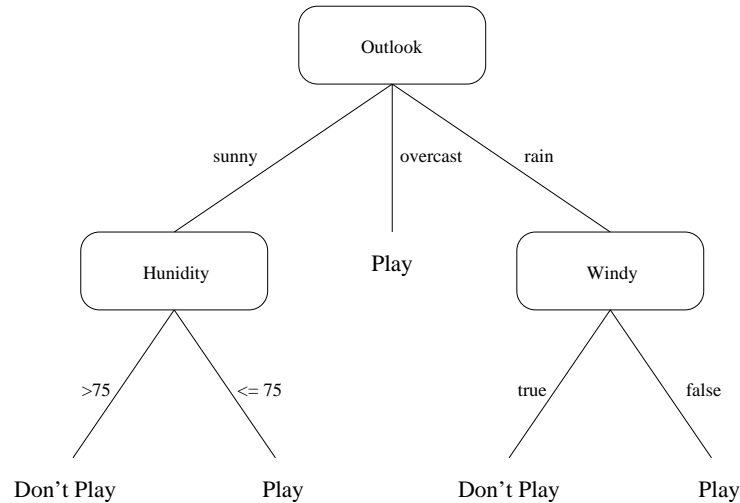
Now that we've shown the method, let us give a brief example (from [Quinlan, 1993]). In table 3 the cases have been grouped on the first attribute to simplify the explanation.

Outlook	Temp (°F)	Humidity (%)	Windy?	Class
sunny	75	70	true	Play
sunny	80	90	true	Don't Play
sunny	85	85	false	Don't Play
sunny	72	95	false	Don't Play
sunny	69	70	false	Play
overcast	72	90	true	Play
overcast	81	78	false	Play
overcast	64	65	true	Play
overcast	81	75	false	Play
rain	71	80	true	Don't Play
rain	65	70	true	Don't Play
rain	75	80	false	Play
rain	68	80	false	Play
rain	70	96	false	Play

Table 3: An example training set

As can be seen, there are cases belonging to different classes, so following from the divide-and-conquer method, it splits up the cases. For this, a test has to be chosen. The way the test is chosen hasn't been discussed yet, but suppose the test is **outlook** with three outcomes: **outlook=sunny**, **outlook=overlook** and **outlook=rain**. The **overcast** group only contains **Play** cases, but the **sunny** and **rain** groups each contain multiple classes. If the **sunny** group is divided by **humidity** ≤ 75 and **humidity** > 75 each of the subsets contain one class. With the **rain** group, **windy=true** and **windy=false** can be used to associate one class with each subset. In figure 32 the corresponding tree is shown.

The choice of the test, as described before seems to be done in a random manner. By just selecting a test and going on from there, a tree can be formed. However, the treebuilding process is not intended to just find such a tree, but also a tree that has predictive power. For that, we need a significant number of cases belonging to each of the leafs. In order to achieve this goal, we could pass all possible trees. However, finding the smallest decision tree consistent with a training set is NP-complete [Hyafil and Rivest, 1976], which would mean that for example in table 3 over 4×10^6 trees would have to

Figure 32: A *decision tree* for the example training set

be examined. We need to have some kind of an algorithm to construct the tree. Most decision tree construction methods are *greedy* algorithms: Once a choice for a particular test has been made, other options for that test are not explored anymore. Within C4.5 the *gain ratio criterion* is used to construct the decision tree.

Gain ratio criterion In this paragraph, the *gain ratio criterion* is explained. First, suppose we have a set T with n outcomes that splits the set into subsets T_1, T_2, \dots, T_n . If we evaluate this test without information coming from the division of the subsets, the only information available for guidance is the distribution of classes in T and its subsets. Now, let us explain the gain ratio criterion, used to determine the choice for a particular test.

Now consider information on a certain case, indicating the outcome of the test for this particular case. We now define for a certain test X

$$split\ info(X) = - \sum_{i=1}^n \left(\frac{|T_i|}{|T|} \right) \times \log_2 \left(\frac{|T_i|}{|T|} \right)$$

The *split info* represents the potential information generated by dividing T into n subsets. The information gain measures the information relevant to classification that arises from the same division. Now we need to know

how the gain is defined: First we introduce S as being any set of cases, let $freq(C_i, S)$ be the number of cases in S that belong to class C_i . Finally, we use $|S|$ to denote the number of cases within S . The theory is built up out of one criterion: The information conveyed by a message depends on its probability and can be measured in bits minus the logarithm to base 2 of that probability. Imagine selecting one case at random from set S of cases and announcing that it belongs to some class C_i . This message has probability

$$\frac{freq(C_j, S)}{|S|}$$

and so the information it conveys

$$-\log_2 \left(\frac{freq(C_j, S)}{|S|} \right) \text{ bits.}$$

To find the expected information from such a message referencing to class membership, we sum over the classes in proportion to their frequencies in S , giving

$$info(S) = - \sum_{i=1}^k \frac{freq(C_j, S)}{|S|} \times -\log_2 \left(\frac{freq(C_j, S)}{|S|} \right)$$

Now, we define $info(T)$ as being the amount of information needed to identify the class of a case in T . When T has been partitioned according to the n outcomes of a test X the expected information requirement is:

$$info_X(T) = \sum_{i=1}^n \frac{|T_i|}{|T|} \times info(T_i).$$

and then the gain of quantity of information by partitioning T in accordance with the test X is defined as

$$gain(X) = info(T) - info_X(T)$$

Now that we have defined the *gain*, we are ready to define the *gain ratio*:

$$gain\ ratio(X) = gain(X) / split\ info(X)$$

The gain ratio criterion states that a test can only be selected if the information gain is at least as great as the average gain over all tests examined.

Now that the method for constructing the tree has been explained, another method comes into play. The trees resulting from the method described above are often too complex and overfit the data. In order to deal with this problem, pruning is used. To simplify the decision trees, one usually replaces one or more subtrees with leaves, in C4.5 another option is included, which is replacing a subtree by one of its branches.

Error-based pruning When the error rate of a tree and its subtrees is known, pruning the tree is simply starting at the bottom of the tree and examine each nonleaf subtree. If replacing this tree with a leaf or with its most frequent used branch leads to a lower error rate, then prune the tree accordingly, remembering that the predicted error rate for all trees that include this one will be effected. The only problem left is how to predict the error-rates. The approach used in C4.5 is to use the training set from which the tree was built. Consider a leaf which is covered by N training cases and E of them are incorrect. The resubstitution error rate for this leaf is E/N . However, the question is what the probability of an error is for the whole set of cases covered by this leaf. Within C4.5 the schema used is that for a given confidence level (CF) the upper limit of the aforementioned error rate can be found from the confidence levels for the binomial distribution, written as $U_C F(E, N)$. Within C4.5 the error rate used is this upper limit. For a more thorough explanation see [Quinlan, 1993]

8.2.2 Reasons for choosing C4.5

Now that we've described how the algorithm works, we are going to explain why we've chosen this algorithm. First of all, this algorithm is able to handle numerical and categorial data, as stated earlier. Secondly, decision trees are very easy to use, you can easily go through the tree and get a classification, also rules can easily be derived from the tree. Finally, this algorithm can give a certain percentage which indicates the chance of being part of a certain class. This can for example be used to determine what belief there is that the current customer is a bad customer and base your decisions on this belief.

8.3 Designing the SalesAnalyst

Now that we've decided on which algorithm to use within the SALESANALYST, we are ready to describe the design. All the features, as listed in section 8.1.1, need to be executed in a sequence somehow, so they can result in the advices that should be given. After the specific tasks have been identified and put in the correct sequence the necessary classes are identified.

8.3.1 UML Activity diagram

We decided to first draw an UML Activity diagram. With this type of diagram, you can easily show how the SALESANALYST determines its advice. The diagram is shown in figure 33.

As can be seen in the figure, the SALESANALYST first determines whether the customer is *known* or *not known*.

If the customer is *not known*, the SALESANALYST determines the chance that this particular customer is a *good* customer. This is determined by means of using the datamining algorithm.

If there is a chance of over or equal to x percent (where $x \leq 100$) of the customer being a *good* customer, the SALESANALYST determines if the customer is wanted as a regular customer, this decision influences other decisions made further on in the SALESANALYST. After that, the tasks to be advised are determined. To help select these tasks, the available information is used to determine which tasks have the highest chance of acceptance. Subsequently, the salesgoals are determined. For this, all the information described above is used. Finally, after the salesgoal has been determined, the SALESANALYST informs the RESOURCEMANAGER how the resources should be allocated.

If there is a chance of less than x percent, the high-risk tasks to be excluded from the bid are determined. If there are any tasks left after the selection, the SALESANALYST continues with the selection of the tasks to be advised, as explained before. If there are no tasks left, the SALESANALYST stops and gives null as an advice.

If the customer is *known*, the type of customer is determined by means of analyzing the data the SALESANALYST has on this customer.

If the analysis showed the current customer is a *good* customer, it determines if the customer is a regular customer or not. If it is regular, it moves on to what has been explained before, namely the determination whether

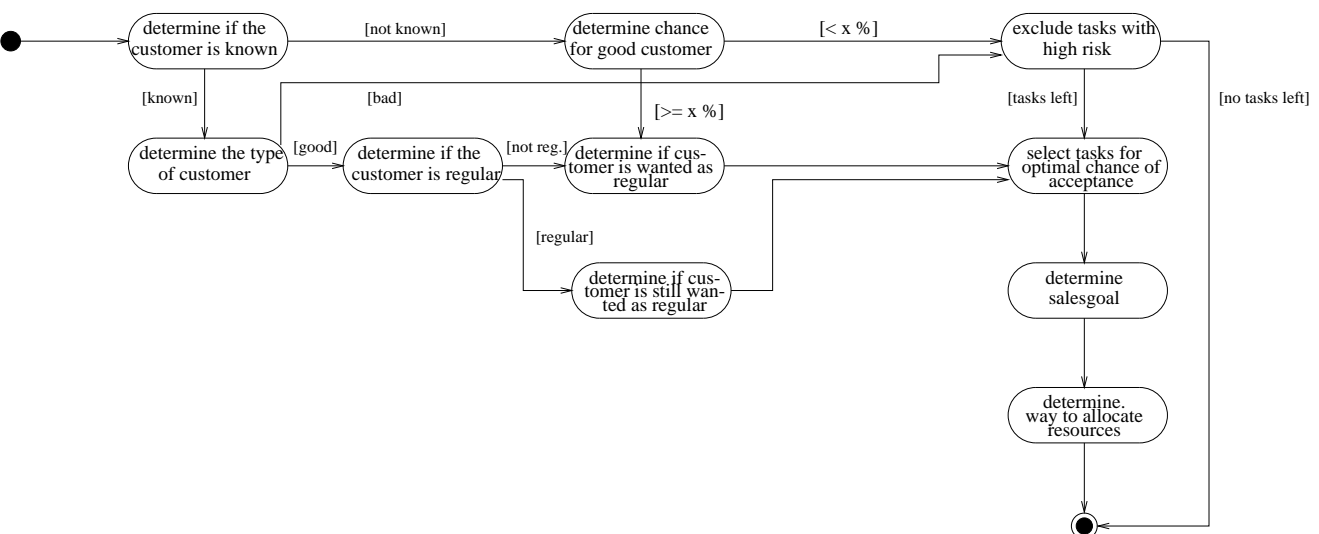


Figure 33: The UML-Activity diagram of the SALESANALYST

the customer is wanted as a regular customer. If the customer is a regular customer, the SALESANALYST determines if the customer is still wanted as a regular customer. After that, it continues with the selection of the tasks, as explained above.

If the analysis shows it concerns a *bad* customer, the SALESANALYST continues with excluding the tasks with a high risk, as presented above.

8.3.2 Describing the tasks

In this section the tasks, as described in the activity diagram, are described.

The *determine if the customer is known* task This task is easy to determine, we just check the database to see if there is an entry in the database of the SALESANALYST for this customer. If there is, the customer is known, otherwise it is not known.

The *determine the type of customer* task This task is used when the customer is already known. The way we want to determine if the known customer is a good customer is by checking the records. If there are two or more records which include a late payment, the customer is considered to be a bad customer and else the customer is considered to be good.

The *determine chance for good customer* task When the customer is not known, we use a datamining technique on the data we saved within this component. The profile includes all information within an RFQ, except for the id's. As mentioned before, we use the *C4.5* algorithm to construct a decision tree. When the tree has been constructed, we take the data in the current RFQ and walk through the tree. After we have encountered a leaf, the percentage of being a bad customer is retrieved.

The *determine if customer is wanted as regular* task If the customer is already known, the task is executed as follows: first of all, determine the load on the tasks included within this RFQ. If this is low enough, determine the expected profit from this customer. The expected profit is determined by getting the resource cost from the RESOURCEMANAGER. If the expected profit is above a certain amount of money, mark the customer

as wanted as a regular customer. In all the other cases, the customer is not wanted as a regular customer.

If the customer is not known, check the percentage that the current customer is a good customer and if it is above a certain percentage y (where $x \leq y \leq 100$), go on with the execution as described above, else, the customer is not wanted as a regular customer.

The *determine if customer is still wanted as regular task* First of all, determine if the customer awards enough bids you issue, this should be according to the definition of a *regular customer* as described before. If this is not the case, the customer is no longer wanted as a regular customer. Otherwise, check the availability of the resources concerning the tasks this customer often sends RFQ's for, if the load is too high skip this customer as a regular customer. If this is not the case, this customer should be kept as a regular customer.

The *exclude tasks with high risk task* To determine the tasks with a high risk, the database of the SALESANALYST is checked for entries concerning a particular task included in the bid. Let's define the percentage of late payments on all bids including this task as z . If z is above some threshold, the task is excluded from the bid. When a customer is known, an extra item is taken into account, namely the payment history of this customer. If the customer regularly doesn't pay for this particular task, the task is also excluded from the advice.

The *select tasks for optimal chance of acceptance task* When the customer is known, first look at the history of the RFQ's and the bids. By looking at whether a bid has been awarded or not, a profile can be determined of the bids that have been awarded. This profile is created by looking in the database and checking for entries from this customer that contain a subset of the current set of tasks. If there are multiple, one has to be chosen, for this we take the one with the highest resource cost, since this would most likely end up in making the highest profit.

If there is no such subset or the customer is not known, scan all entries and check to see if there is a subset of the current task which has been awarded. If there are multiple, choose the one with the highest possibility of acceptance (percentage of bids with exactly these tasks which have been

awarded of the total number of bids) is taken. If none of the entries contain a subset of the tasks, then bid on all the tasks.

The *determine salesgoal* task The task of the determination of the salesgoal is done in the following way: first, the RESOURCEMANAGER is consulted on the average load on the resources concerning the tasks that have been selected. If the load, l is high ($75 \leq l \leq 100$), the salesgoal is *not eager to sell*. If the load is low ($0 \leq l \leq 25$) the salesgoal is *eager to sell*. Finally if the load is not low or high ($25 < l < 75$), there are several possibilities: If the customer is wanted as a regular customer and isn't a regular customer yet, the salesgoal is *bind customer*. If the customer is a regular customer and needs to remain one, the *maintain as regular customer* salesgoal is passed. The *reduce risk* salesgoal is passed, when the current customer is a bad customer. Finally, the *none* salesgoal is passed in all other cases.

The *determine way to allocate resources* task In order to accomplish this task, first determine the chance for acceptance. The chance is determined in the following way: if the customer is known and there is a history concerning these specific tasks, the chance is the percentage of the bids (contained precisely these tasks) which has been awarded. If there is no history involving such tasks, take the average of bids awarded by this customer.

If the customer is not known, look at entries involving bids on exactly these tasks, and take the percentage of these bids that have been awarded. If there is no such entry, determine how many of the bids are usually awarded and take this percentage.

After the chance has been determined, the salesgoal is taken into account. The chance of acceptance is taken as a base. In case the salesgoal is *eager to sell*, the SALESANALYST lowers the base, since it wants to sell the product, and with the lower base it can bid on more RFQ's containing this task. When the salesgoal is *not eager to sell*, the base is increased. In case the salesgoal is *reduce risk*, the base is also lowered since the customer isn't very reliable. When the salesgoal is *bind customer* the base is increased because the customer needs to get faith in this SUPPLIERAGENT, so the resources should be available. The same is the case for a *maintain as regular customer* salesgoal. Finally, for the *none* salesgoal, the base is not adjusted.

8.3.3 Identifying the classes and the interfaces

Now that we've identified all the tasks we are ready to define the classes. The UML Class diagram is shown in figure 34.

As can be seen, we've identified four classes, namely SALESANALYST, DATAPROCESSING, DATABASE and C45ALGORITHM. The SALESANALYST models the component, and has the general supervision of the advice to be passed. To do the process of getting information from the data, the DATAPROCESSING class is introduced. Within this class no decisions are taken, only data from the market are processed into the correct format. In order to be able to maintain the necessary data, the DATABASE class is used. Finally, the C45ALGORITHM class includes the *C 4.5* datamining algorithm, used to derive the information as explained before. We've chosen to do it this way because if you split the system up like this you abstract the way the data is stored and the way the data is used from each other.

The SALESANALYST has exactly one DATABASE and every DATABASE belongs to exactly one SALESANALYST. It could be that a DATABASE is shared by multiple SALESANALYST components, but this would mean problems with different Threads, so we decided to model it like this. Another problem with one DATABASE for all components is that the control of the updating of the DATABASE would be difficult, since only one SALESANALYST should update the DATABASE. Figuring out which one this should be increases the complexity of the SALESANALYST component as a whole. If performance would be more import, it could easily be remodeled as one DATABASE for all SALESANALYST components. The DATAPROCESSING class owns exactly one DATABASE (which is the same as the one owned by the SALESANALYST and the DATABASE belongs to exactly one DATAPROCESSING object. Finally, the DATAPROCESSING object owns one C45ALGORITHM object and that one belongs to exactly one DATAPROCESSING object. Only one C45ALGORITHM is used per DATAPROCESSING class because there is no intention to run several datamining processes at the same time.

8.4 Implementation

Before starting to implement, the issue of testing the SALESANALYST is addressed. Since the SALESANALYST we designed requires a more intelligent customer agent than currently present in the MAGNET system, it's difficult

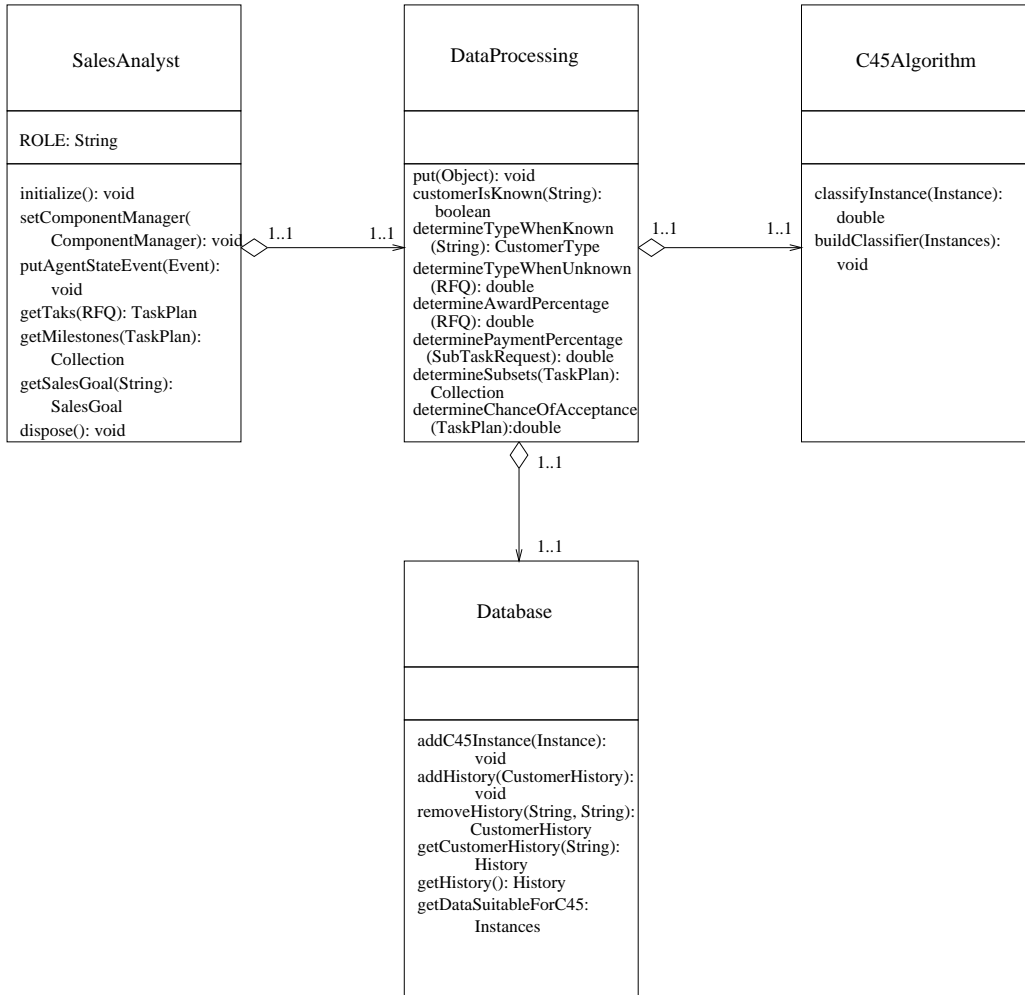


Figure 34: The UML class diagram of the SALESANALYST

to test and draw conclusions about all the features included in the SALESANALYST. For example, the datamining algorithm is used to determine the profile of good and bad customers but the current MAGNET system doesn't support different customer agents. Ofcourse it's possible to adjust the system so there can be multiple customers, however, by creating a profile for good and bad customers manually, it is hard to do objective testing. Making a sound conclusion based on the results following from the aforementioned tests is almost impossible since the samples were generated with the constructed SALESANALYST in mind.

8.4.1 Feeding information to the C4.5 Algorithm

Now that we need to implement the DATABASE class, we need to determine what information can be taken by the C4.5 algorithm. First of all, all cases have to be similar and can't be an array of things, so the tasks each have to be contained in a seperate entry. Because of this, the table needs to be updated every time a new type of task comes in, and every entry contains a *no* to model that the task is not part of the RFQ (except for the newly arrived RFQ ofcourse, it contains a *yes* for this task). Just adding a date to the cases wouldn't have much meaning and wouldn't be usefull to learn from, that's why we've decided to derive the time of day, the day of the week and the month from the date. Another issue is how to model the timewindows in the instances. Since probably not all tasks are in the RFQ it wouldn't be possible to only include the information from the tasks included in the RFQ, since the algorithm needs cases with the same number of attributes. If we would use the timewindows we would have to use a special timewindow for the task that is not present in the RFQ. However, learning from this would be very difficult and may mess up the learning because it's not usefull to learn about timewindows from tasks that are not present. Because of this, we've decided to derive information from the timewindows and use this to learn. The information we derive is the following: We ask the RESOURCEMANAGER the avarage duration of the execution of the specific task and determine how the timewindow relates to this duration. We define this info as follows (n is the number of tasks):

$$timewindow_info = \left(\sum_{i=1}^n \frac{timewindow_size(i) - average_duration(i)}{average_duration(i)} \right) / n.$$

Another piece of information we extract from the timewindows is the

standard deviation for the `timewindow_info`, defined in the following way:

$$t_w_standard_deviation = \left(\sum_{i=1}^n | (timewindow_size(i) - average_duration(i)) - timewindow_info | \right) / n$$

Finally, we include information on the time (in seconds) between the arrival of the bid and the `bid_deadline`, `earliest_offer_time` and `earliest_consider_time`. This would be information more suitable for learning, since you can for example see how eager a customer is to get the job done.

Now the entries in the table for the C4.5 algorithm looks like this:

- **class**, where the possible values are *good* and *bad*.
- **task_1**, where the possible values are *yes* and *no*.
- •
•
- **task_n**, where the possible values are *yes* and *no*.
- **timewindow_info**, where the values can be those of a double.
- **t_w_standard_deviation**, where the values can be those of a double.
- **time_of_issue** where the possible values are *night*, *morning*, *afternoon* and *evening*.
- **day_of_issue** where the possible values are the days within the week: *Monday*, *Tuesday*, *Wednesday*, *Thursday*, *Friday*, *Saturday* and *Sunday*.
- **month_of_issue** where the values are the possible months within the year, so *January*, *February*, *March*, *April*, *May*, *June*, *July*, *August*, *September*, *October*, *November* and *December*.
- **bid_deadline** where the value can be all the values of a double.
- **earliest_offer_time** where a double models the possible values.
- **earliest_consider_time** again a double models the possible values.

8.5 Running experiments

Now that we've designed the SALESANALYST, we need to run experiments to determine whether our implementation works. There are a number of variables we can vary (since they've already been treated, they are not explained in great detail): The variable x , with which the percentage resulting from the C4.5 algorithm needed for an unknown customer to be treated as a good customer is modeled. The use of varying this parameter is to investigate whether taking a risk (having a lower value for x) pays off. The variable y used to model the percentage from the C4.5 for being considered wanted as a regular customer. Varying y tells us what the effect of having different amounts of regular customers is. z can be varied too, it acts as a minimum change, above which the tasks are excluded when a customer is a *bad* customer. Finding the value for z so that the trade-off between number of bad payments and awards is optimal, this is essential for a good SALESANALYST.

8.5.1 Setup

The experiments have been set up in the following way: We've constructed a program which can be used to simulate a certain number of *customer agents*. When the program is initialized, a certain percentage of the agents being a *bad customer* is given as a parameter. There are various other parameters that can be varied, this includes: The percentage of bad payments the bad customers do, the total number of customers, the number of tasks per bid and the total number of different types of tasks. The aforementioned parameters are not varied during testing, but are given values which are realistic in our opinion. After the program has been initialized the *supplier agents* are constructed. In order to be able to measure the performance of our algorithm, a RANDOMSALESANALYST has been constructed to act as a bias. The RANDOMSALESANALYST uses a simple algorithm to select the tasks from the RFQ: a random number is generated, if the number is above a certain threshold, the task is added to the bid. When constructing the *supplier agents* the total number active in the MARKET can also be varied. All of the *supplier agents* use the same PRICEMANAGER, which is the fixed markup PRICEMANAGER (see [Schoolcraft, 2002]). The markup was kept constant during the experiments and was the same for all the *supplier agents*. All of the agents use the same RESOURCEMANAGER too, this component returns every task as being available and returns the largest possible time window.

After 1400 RFQ's have been generated, the results of the different types of agents are compared. Comparing the performance of the agents has been done with several different measurements: First of all, the number of awards per agent have been compared. The number of bad payments per agent is another type of measurement that has been used. Since the binding of the *customer agents* to the *supplier agents* is currently not supported by the *customer agents*, this performance isn't measured. In the next section, the results are shown.

8.5.2 Results

The z parameter First of all, the z parameter has been given a set of different values: $z = \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1\}$, the other parameters have been kept constant. In figure 35 the effect of changing the z parameter on the amount of awards is shown, the influence on the amount of bad payments received is shown in the same figure. In order for the figure to be clear, the results of the RANDOMSALESANALYST are not shown in this figure.

As can be seen, increasing the z parameter results in getting more awards, however the amount of bad payments increases rapidly when z is increased. When the z parameter reaches the value 0.2, increasing it to 0.3 leads to a significant increase in the amount of bad payments. Because of the aforementioned results, the value 0.2 seems to be the most satisfying value. In figure 36 the performance of the DMSALESANALYST with $z = 0.2$ against the RANDOMSALESANALYST is shown. Besides this, the same has been done for the value 0.1, which is shown in figure 37. The DMSALESANALYST clearly outperforms the RANDOMSALESANALYST. First of all on every interval it receives significantly more awards compared to the RANDOMSALESANALYST. Secondly, the amount of *bad* payments received is significantly better after 300 RFQ's have been issued. The results in the beginning are worse because the DMSALESANALYST receives a lot more awards, resulting in a higher chance on a *bad* payment, and the DMSALESANALYST needs some time to determine the profiles of the customers.

The y parameter Varying the y parameter in our current test environment doesn't provide us with usefull information. This is because the current *customer* doesn't support preferences for particular *supplier agents*. Because of this, a lower y will lead to more awards because more *customers* are wanted

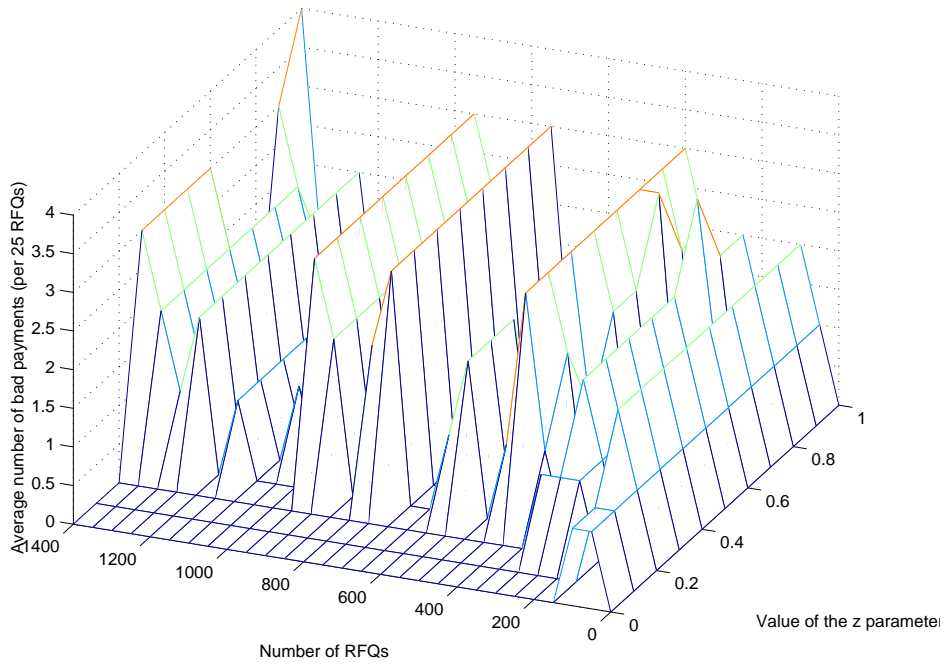
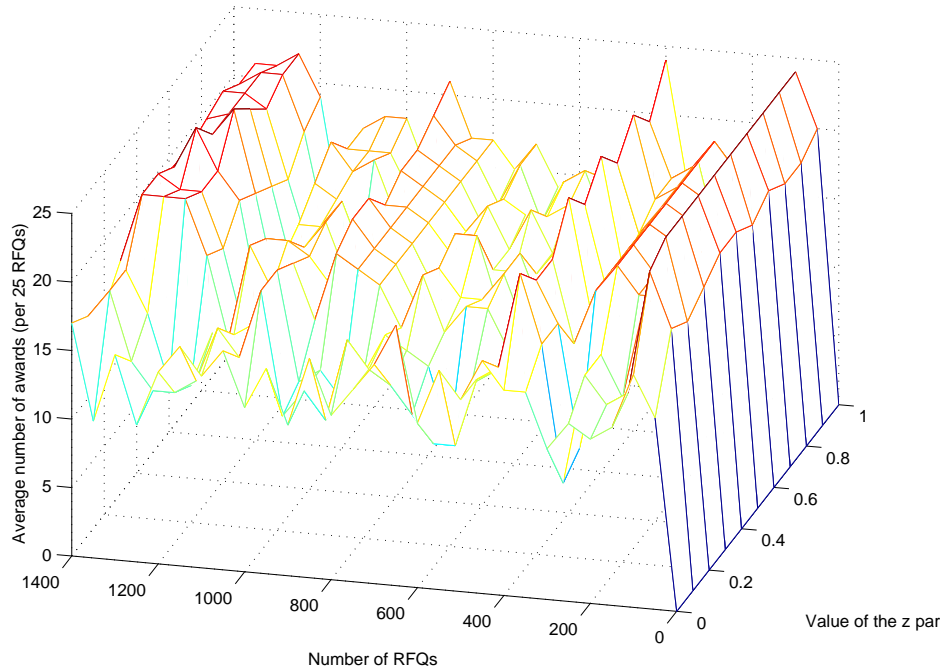


Figure 35: Experiment with 1400 RFQ's and $x=0.90$, $y=0.98$ and z variable

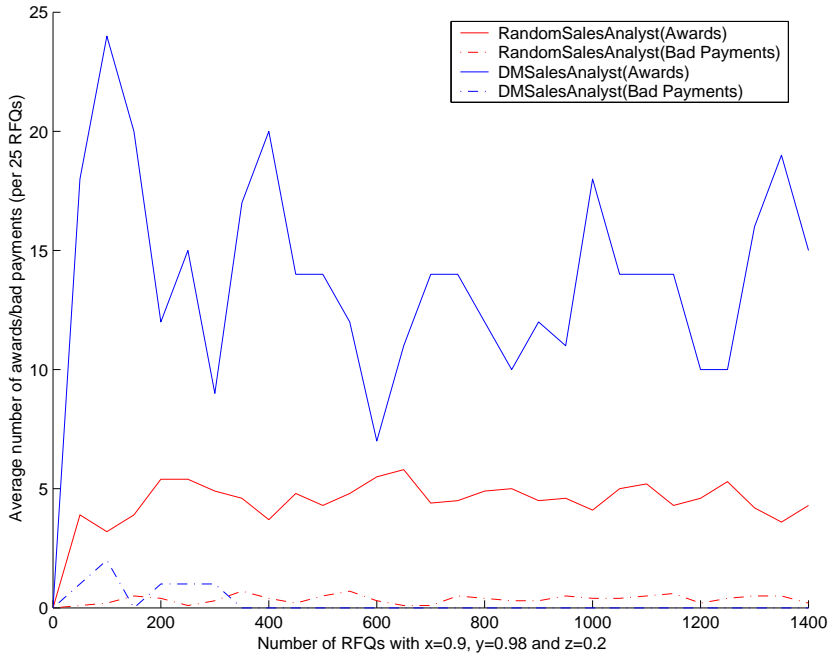


Figure 36: Experiment with 1400 RFQ's and $x=0.90$, $y=0.98$ and $z=0.20$

as regular and will be offered a better price. Some brief results are shown in figure 38.

As can be seen the results are almost identical, some slight variations can be seen but those are so small that no sound conclusion can be drawn.

The x parameter Varying the x parameter is again not useful in our test environment. Therefore, the results are not shown. This is because the profiles and bids of the customers are randomly generated and this will not result in particular profiles for *good* and *bad* customers.

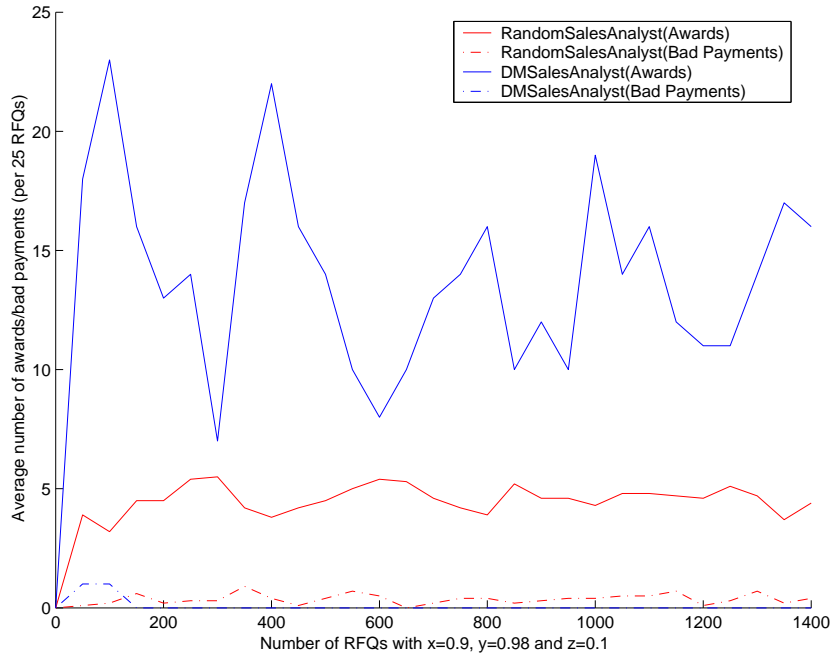


Figure 37: Experiment with 1400 RFQ's and $x=0.90$, $y=0.98$ and $z=0.10$

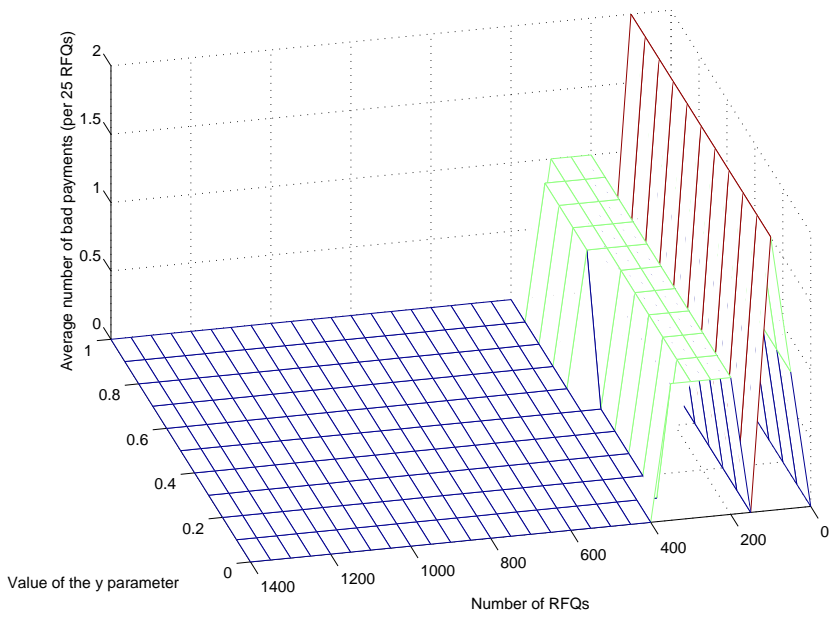
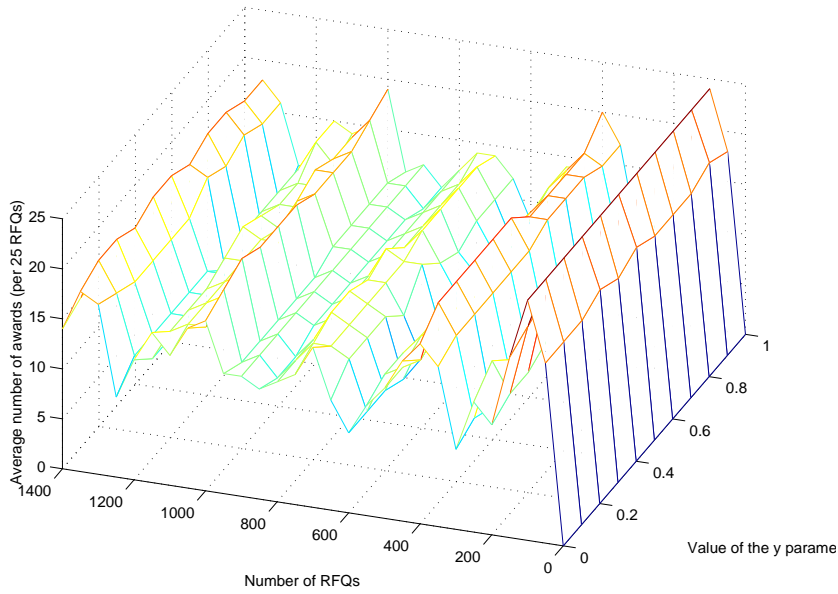


Figure 38: Experiment with 1400 RFQ's and $x=0.90$, $z=0.20$ and y variable

9 Conclusions and Future Work

The `SUPPLIERAGENT` that has been discussed in this paper has been fully and successfully implemented. The implementation provides a good framework to be able to do research in the individual components of the `SUPPLIERAGENT` and succeeds in the goal to create an easily configurable `SUPPLIERAGENT`. The `SUPPLIERAGENT` designed is very portable thanks to the separate `COMMUNICATIONMANAGER` component. In case the `SUPPLIERAGENT` wants to interact on a different market, it just needs a `COMMUNICATIONMANAGER` able to communicate with this market. Another advantage of our design is that the components are relieved from the burden of maintaining their own administration because the `AGENTSTATE` takes care of this. Third, the fact that the components are very independent because of the component-based design is also an advantage. The framework has been successfully tested in several ways. First of all, the `SALESANALYST` that has been introduced in this paper was very easy to implement thanks to the framework. The only thing that needed to be done was extending the dummy class with the algorithm and subscribing to the appropriate events. These events are needed for the algorithm to get the necessary information and to be able to measure the performance of the algorithm. Another example of the usefulness of the framework is the implementation of the several `PRICEMANAGERS` by Schoolcraft. Finally, an implementation of the `RESOURCEMANAGER` has been introduced by Hawkins.

A lot of future work is to be done, first of all, the current framework should be extended so it can interact with the `MARKET` of the `MAGNET` system. There is currently only an implementation of the `COMMUNICATIONMANAGER` that communicates directly to the *customers*. Another extension for the framework is to have even more independent components; a proposal has been done by Collins to have no predefined sequence in how the bid is constructed. All components have access to the `RFQ` and the bid that has been constructed at that point and can change it if it's available. Further, the `BIDMANAGER` should be further developed, in the current implementation the only thing the `BIDMANAGER` does is collect all the advices of the individual components and put them in a bid. A proposal has been done in [Schoolcraft, 2002] to have several different implementations of the components. First is the customer-relationship management approach. In this scenario, the `SalesAnalyst` influences the actions of the `PriceManager` and `ResourceManager` in order to create bids whose timing and price attract and

keep the loyalty of the most desirable customers. Another method is the resource-centric model. The agent works at scheduling resources and setting prices so as to optimize its schedule and reduce waste. A third strategy is to seek out the best prices the market will bear. The agent does this by experimenting with prices and seeing how much profit it can get away with. Resources can be managed any way the user likes. Customer information can be ignored, or used in a simple form, such as not bidding for known delinquent customers. Several of these scenarios have been developed already, but should be further examined. We've been doing work in the first scenario.

The results that have been found while running tests with the SALESANALYST introduced in section 8 are very preliminary. The testing has been done with strictly random data and this leads to the generating of RFQ's belonging to a particular customer. Because of this randomness the SALESANALYST can hardly learn from the history, the only real feature that can be tested is whether the amount of bad payments decreases in time. The results for this feature are very promising.

To be able to thoroughly test the SALESANALYST in the future we either need to have a more intelligent customer, which actually binds to suppliers and has a profile of it's own, or real life data. The problem with designing a more intelligent customer is that this agent is designed to test our SALESANALYST and the results will not be independent. Obtaining real life data is difficult, since this information is mostly kept private by the companies. An attempt has been made to get data from the international shipping domain, but this failed. Another option is to use the evolutionary framework proposed by [Babanov *et al.*, 2002b], this approach could be used to determine the optimal settings of the DMSALESANALYST, and see if this DMSALESANALYST would survive between different types of SALESANALYSTS.

10 Acknowledgements

The MAGNET project is supported by the National Science Foundation, under awards NSF/IIS-0084202 and NSF/EIA-9986042.

This project wouldn't have been possible without the help of a lot people. First of all we would like to thank Maria Gini for guiding us through the project and always being there when we needed assistance. We would also like to thank her for inviting us at the University of Minnesota without knowing much about us. Further we would like to thank Catholijn Jonker for giving us the opportunity to study abroad and being our supervisor at the Vrije Universiteit. We owe thanks to John Collins for helping us with solving problems on a more detailed level of the MAGNET system. Of course we would like to thank everybody participating in the MAGNET group, especially Anne Schoolcraft who implemented the PRICEMANAGER we needed for our experiments and Yelena Kryzhnyaya who provided us with the classes needed to test our implementation. We would also like to thank our parents for supporting us during our project and stay in the United States. Finally, we would like to give special thanks to our old, but faithful *Dodge '91 Dynasty* for always taking us where we needed to go.

References

- [Babanov *et al.*, 2002a] Alex Babanov, John Collins, and Maria Gini. Risk and expectations in a-priori time allocation in multi-agent contracting. In *Proc. of the First Int'l Conf. on Autonomous Agents and Multi-Agent Systems*, Bologna, Italy, July 2002.
- [Babanov *et al.*, 2002b] Alexander Babanov, Wolfgang Ketter, and Maria Gini. Presentation: An evolutionary framework for large-scale experimentation in multi-agent systems. Toward an Application Science: MAS Problem Spaces and Their Implications to Achieving Globally Coherent Behavior, Bologna, Italy, 2002.
- [Bauer *et al.*, 2001] Bernhard Bauer, Jrg P. Mller, and James Odell. Agent uml: A formalism for specifying multiagent interaction. In Paolo Ciancarini and Michael Wooldridge, editors, *Agent-Oriented Software Engineering*, pages 91–103, Berlin, 2001. Springer-Verlag.
- [Bounsaythip and Rinta-Runsala, 2001] Catherine Bounsaythip and Esa Rinta-Runsala. Overview of data mining for customer behavior modeling. Technical Report TTE1-2001-18, VTT Information Technology, Espoo, Finland, June 2001.
- [Brazier *et al.*, 1998] F.M.T. Brazier, C.M. Jonker, and J. Treur. Principles of compositional multi-agent system development. In J. Cuen, editor, *Proceedings of the 15th IFIP World Computer Congress, WCC'98, Conference on Information Technology and Knowledge Systems, IT&KNOWS'98*, pages 347–360, 1998.
- [Chavez and Maes, 1996] Anthony Chavez and Pattie Maes. Kasbah: An agent marketplace for buying and selling goods. In *Proc. of the First Int'l Conf. on the Practical Application of Intelligent Agents and Multi-Agent Technology*, London, UK, April 1996.
- [Choi and Liu, 2001] Samuel P. M. Choi and Jiming Liu. A dynamic mechanism for time-constrained trading. In *Proc. of the Fifth Int'l Conf. on Autonomous Agents*, pages 568–575, 2001.
- [Collins and Gini, 2001a] John Collins and Maria Gini. Bid evaluation for coordinated tasks: an integer programming formulation. In *IJCAI-2001 Workshop on Economic Agents, Models, and Mechanisms*, August 2001.

- [Collins and Gini, 2001b] John Collins and Maria Gini. A testbed for multi-agent automated contracting. In *IJCAI-2001 Workshop on Artificial Intelligence and Manufacturing*, August 2001.
- [Collins *et al.*, 1998] John Collins, Ben Youngdahl, Scott Jamison, Bamshad Mobasher, and Maria Gini. A market architecture for multi-agent contracting. In *Proc. of the Second Int'l Conf. on Autonomous Agents*, pages 285–292, May 1998.
- [Collins *et al.*, 2000] John Collins, Corey Bilot, Maria Gini, and Bamshad Mobasher. Mixed-initiative decision support in agent-based automated contracting. In *Proc. of the Fourth Int'l Conf. on Autonomous Agents*, pages 247–254, June 2000.
- [Collins *et al.*, 2002] John Collins, Wolfgang Ketter, Maria Gini, and Bamshad Mobasher. A multi-agent negotiation testbed for contracting tasks with temporal and precedence constraints. *Int'l Journal of Electronic Commerce*, 2002.
- [Faratin *et al.*, 1997] Peyman Faratin, Carles Sierra, and Nick R. Jennings. Negotiation decision functions for autonomous agents. *Int. Journal of Robotics and Autonomous Systems*, 24(3-4):159–182, 1997.
- [Hunt *et al.*, 1966] E.B. Hunt, J. Marin, and P.J. Stone. *Experiments in Induction*. Academic press, New York, 1966.
- [Hyafil and Rivest, 1976] L. Hyafil and R.L. Rivest. Constructing optimal binary decision trees is np-complete. In *Information Processing Letters* 5, pages 15–17, 1976.
- [Karacapilidis and Moraïtis, 2001] Nikos Karacapilidis and Pavlos Moraïtis. Intelligent agents for an artificial market system. In *Proc. of the Fifth Int'l Conf. on Autonomous Agents*, pages 592–599, 2001.
- [Kinny and Georgeff, 1997] D. Kinny and M. Georgeff. Modeling and design of multi-agent systems. In M. J. Wooldridge J. P. Miller and N. R. Jennings, editors, *Intelligent Agents III*, Lecture Notes in Artificial Intelligence. Springer, Berlin, 1997.
- [Loritsch, 2001] B. Loritsch. Developing with Apache Avalon. Apache Software Foundation, 2001.

- [Mobasher *et al.*, 1996] B. Mobasher, N. Jain, E. Han, and J. Srivastava. Web mining: Pattern discovery from world wide web transactions. Technical Report 96-050, University of Minnesota, Department of Computer Science, Minneapolis, MN, September 1996.
- [Nisan, 1999] Noam Nisan. Bidding and allocation in combinatorial auctions. In *1999 NWU Microeconomics Workshop*, 1999.
- [Odell, 1999] James Odell. Objects and agents: Is there room for both? Distributed Computing, november 1999.
- [Padgham and Winikoff, 2002] L. Padgham and M. Winikoff. Prometheus: A methodology for developing intelligent agents, 2002.
- [Quinlan, 1993] J. Ross Quinlan. *C4.5 Program for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, California, 1993.
- [Sandholm, 1996] Tuomas W. Sandholm. *Negotiation Among Self-Interested Computationally Limited Agents*. PhD thesis, Department of Computer Science, University of Massachusetts at Amherst, 1996.
- [Schoolcraft, 2002] Anne Schoolcraft. Price manager. Master's thesis, University of Minnesota, 2002.
- [Shehory, 1998] O. Shehory. Architectural properties of multi-agent systems, 1998.
- [Srivastava *et al.*, 2000] Jaideep Srivastava, Robert Cooley, Mukund Deshpande, and Pang-Ning Tan. Web usage mining: Discovery and applications of usage patterns from web data. *SIGKDD Explorations*, 1(2):12–23, 2000.
- [Sun Microsystems inc., 1995] Sun Microsystems inc. Threads: Doing two or more tasks at once. <http://java.sun.com/docs/books/tutorial/essential/threads/>, 1995.
- [Sycara and Pannu, 1998] Katia Sycara and Anandeeep 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.

- [Szyperski, 1998] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley/ACM-Press, 1998.
- [Tsvetovatyy *et al.*, 1997] Maxsim Tsvetovatyy, Maria Gini, Bamshad Mobasher, and Zbigniew Wieckowski. MAGMA: An agent-based virtual market for electronic commerce. *Journal of Applied Artificial Intelligence*, 11(6):501–524, 1997.
- [Wang *et al.*, 2000] Shi Wang, Wen Gao, Jintao Li, Tiejun Huang, and Hui Xie. Web clustering and association rule discovery for web broadcast. In *Web-Age Information Management*, pages 227–232, 2000.
- [Wellman and Wurman, 1998] Michael P. Wellman and Peter R. Wurman. Market-aware agents for a multiagent world. *Robotics and Autonomous Systems*, 24:115–125, 1998.
- [Wooldridge and Jennings, 1999] M. Wooldridge and N. R. Jennings. Software engineering with agents: pitfalls and pratfalls, May/June 1999.
- [Wurman *et al.*, 1998] Peter R. Wurman, Michael P. Wellman, and William E. Walsh. The Michigan Internet AuctionBot: A configurable auction server for human and software agents. In *Second Int'l Conf. on Autonomous Agents*, pages 301–308, May 1998.
- [Zaki *et al.*, 1996] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Wei Li, and Mitsunori Ogihara. Evaluation of sampling for data mining of association rules. Technical Report TR617, University of Rochester, Computer Science Department, Rochester, NY, 1996.

A Paper for AAMAS '02 conference

This section includes the paper that has been submitted and accepted for the Agent-Oriented Information Systems workshop at the first international joint conference on Autonomous Agents and Multiagent Systems which took place in July 2002, Bologna, Italy.

Design of supplier agents for an auction-based market*

Stefan Botman, Mark Hoogendoorn,
Vasile Bud, Ashutosh Jaiswal, Steve Hawkins,
Yelena Kryzhnyaya, Janice Pearce, Anne Schoolcraft, Espen Sigvartsen,
John Collins, and Maria Gini

University of Minnesota

Abstract. We are interested in supporting multi-agent contracting in which customer agents solicit the resources and capabilities of other self-interested supplier agents in order to accomplish their goals. Goals may involve the execution of multi-step tasks in which different tasks are contracted out to different suppliers.

In this paper we focus on the design of supplier agents. The agents are designed to operate in the context of the MAGNET (Multi AGent NEgotiation Testbed) system, but the design could easily be adapted to other situations in which agents interact through a market infrastructure.

MAGNET supplier agents can register their capabilities with the market, be notified of open and relevant requests for quotations, and submit bids that specify which tasks they are able to undertake, when they are available to perform those tasks, and at what price. Supplier agents attempt to maximize the value of their resources.

The paper describes the detailed design of supplier agents and presents preliminary experimental results.

1 Introduction

Online marketplaces offer benefits to both buyers and sellers. For buyers, a marketplace can significantly ease the process of searching for and comparing providers, while for sellers, marketplaces provide access to much broader customer bases. The major challenge in extending currently available online marketplaces comes from the necessity to go beyond simple buying and selling. A realistic system needs to incorporate time constraints, to enforce deadlines, to interact with a highly distributed web of suppliers with different capabilities and resources, to interact over long periods of time through the completion of the contracted work, and to deal with failures in contract execution.

The proliferation of business-to-business portals such as CommerceOne (www.commerceone.com) and VerticalNet (www.verticalnet.com) shows the need and industry demand for value-added services such as security, match-making, and trusted intermediaries. A framework which can successfully address

* Work supported in part by the National Science Foundation, awards NSF/IIS-0084202 and NSF/EIA-9986042

the full spectrum of the requirements mentioned above needs to provide support for contracting activities among participants, as well as provide support for automated agents that act on behalf of human participants.

In order to model these features the MAGNET (Multi-Agent Negotiation Testbed) system has been designed at the University of Minnesota [Collins *et al.*, 1998]).

2 The MAGNET system

The MAGNET architecture provides a framework for secure and reliable commerce among self-interested agents. What makes MAGNET unique is its ability to support negotiation of contracts for tasks that have temporal and precedence constraints [Collins *et al.*, 2002]. MAGNET shifts much of the burden of market exploration, auction handling, and preliminary decision analysis from human decision-makers to a network of heterogeneous agents.

2.1 A Motivating Example

For example, imagine that we need to construct a house. Figure 1 shows the tasks needed to complete the construction. The tasks are represented in a *task network* where links indicate precedence constraints. The first decision we must make is how to sequence the tasks in the *Request for Quotes* (RFQ) and how much time to allocate to each of them. For instance, we could reduce the number of parallel tasks, allocate more time to tasks with higher variability in duration or to tasks for which there is a shortage of laborers, or to allow more slack time. We assume that suppliers are more likely to bid, and to submit lower-cost bids, if given greater flexibility in scheduling resources [Babanov *et al.*, 2002], so time windows in the RFQ might overlap. A sample RFQ is shown in Figure 1. Note that the time windows in the RFQ do not need to obey the precedence constraints; the only requirement is that the accepted bids obey them.

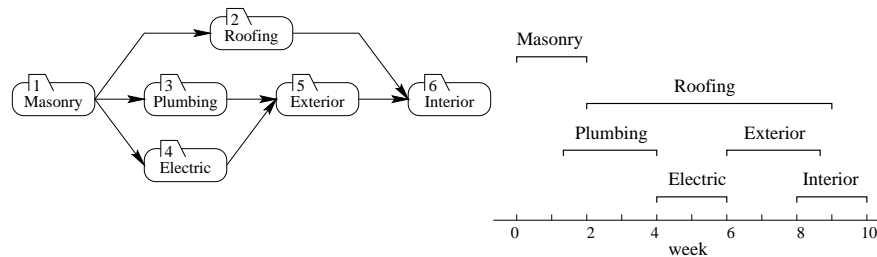


Fig. 1. A task network example and a corresponding RFQ.

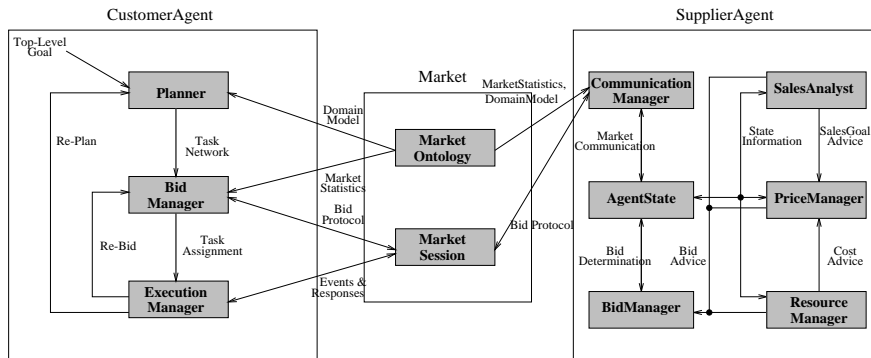


Fig. 2. The MAGNET architecture.

2.2 The MAGNET Architecture

The architecture of MAGNET is illustrated in Figure 2. The MAGNET system consists of *supplier* agents, *customer* agents, and a *market*. The supplier agent has resources to offer, while the customer agent has resource and/or service needs.

This is a schematic outline of the main interactions among agents.

- A customer agent issues a *Request for Quotes* (RFQ) which specifies tasks, their precedence relations, and a time line for the bidding process. For each task, a time window is specified giving the earliest time the task can start and the latest time the task can end.
- Supplier agents submit bids. A bid includes a set of tasks, a price, a portion of the price to be paid as a non-refundable deposit, and estimated duration and time window data that reflect supplier resource availability and constrain the customer’s scheduling process.
- The customer agent decides which bids to accept. Each task needs to be mapped to exactly one bid (i.e. no free disposal [Nisan, 1999]), and the constraints of all awarded bids must be satisfied in the final work schedule. In MAGNET the customer can choose from a collection of winner-determination algorithms (A^* , IDA^* , simulated annealing, and integer programming).
- The customer agent awards bids and specifies the work schedule.

3 Architectural Design Principles

We now briefly outline the design principles behind MAGNET. We have adopted several design principles that make it easy to plug components together and to reconfigure the system. Examples are:

1. The system is written in Java and has been tested on multiple platforms. This makes it easy to adopt on whatever platform you happen to be using.

2. The system is organized into a set of components, and a set of systems that can be constructed from various subsets of components. Each system is constructed to serve a particular experimental purpose.
3. All the major behavioral modules are written as abstract classes, with (potentially) multiple implementations that can be “plugged in” to implement a particular behavioral variant.
4. Virtually every feature of the system is selectable and configurable from a configuration file, and many of these can be viewed and changed from a user interface. This includes the choice of behavioral plug-ins.
5. The interface between the agents and the Market is also abstracted. This allows connection with multiple types of markets (such as one that looks up price and availability info from a catalog or timetable) and through multiple communications protocols.
6. Much of the activity of the agent is agenda-driven, and development and maintenance of the agenda is an important activity in its own right. Agenda items can select plug-ins, update configuration details, evaluate options, interact with the market or other agents, update the agenda, and record results.
7. A pervasive logging and data collection system allows for both detailed examination of behavior and the generation of experimental data. The level of logging detail may be independently configured for different modules, and the various logging levels have well-defined meanings.

The MAGNET market exists as an EJB and interacts with customer and supplier agents through the use of the SOAP services described earlier. Market sessions exist also as EJBs and are created and accessed through the use of market-related SOAP services. Session persistence is addressed through the use of entity beans, which are persistent as needed through the use of an application server’s database. Session-related EJBs are used by the RFQ, bid submission, and bid award SOAP services.

3.1 Security

The current MAGNET implementation does not have robust security, except for the rudimentary security provided by Java’s sandbox model. The security needs for the system can be classified into three main components: (1) authentication, (2) authorization, and (3) non-repudiation.

Authentication is the process of identifying an entity, based on the information provided by it and verified by the system. In MAGNET, agents join and leave as needed to carry out transactions with other agents. The agents need to be authenticated before they join the system. We plan to carry out this process through the market. Absence of a secure authentication mechanism leaves the potential for rogue agents sneaking into the system.

An agent operates on behalf of its principal, which may be a person, a corporation or some other physical entity. Authenticating an agent would mean

authenticating the principal indirectly. Public Key Infrastructure (PKI) and Kerberos are two acknowledged methods used for authentication. The first involves using digital certificates from certificate authorities, which are presented by an entity to the authenticating system each time its identity needs to be established. Kerberos involves using unique keys, called tickets, to exchange secure messages between two entities on an open network. The decision on which system to use requires analyzing the computational power needed to perform encryption of the channel.

Authorization is the process of granting or denying access to system objects to an entity based on its identity. This step is usually preceded by authentication. In the MAGNET system, agents utilize market resources to exchange messages with other agents, place and receive bids etc. In order to use the market resources, the agents need to have proper authorization. This can be done based on the origin of the code (signer) or the user executing the agent code.

Non-repudiation is a property achieved through cryptographic methods which prevents an entity from denying having performed a particular action related to a set of data [OECD, 1997]. What this means is that an agent would not be able to deny an operation after committing it. This is a desirable property to maintain trust in the system. If the first two components are executed properly, non-repudiation would just require the use of logs for these transactions.

3.2 Recovery of Agent State

To increase reliability, MAGNET agents periodically save their state in a database, which allows for recovery in case of a crash.

The database we selected is JDBM¹ (Java Data Base Manager). JDBM offers persistent storage, and is a relatively fast and simple database engine. All updates are transactionally safe. JDBM offers scalable data structures, such as Hash trees and B+ trees. All operations in the database have ACID properties, and the database uses a transaction log and can perform a recovery in case of a crash.

The JDBM database engine can be configured in different ways. For example, by choosing to synchronize the transaction log less frequently, the database performs better. The trade-off is that a recovery will be more time consuming. This feature is interesting when trying to keep the overhead as low as possible, as it is the case in MAGNET. In our current implementation, the agent state is saved between the time- and resource-consuming processes, such as the search algorithms for winner-determination.

4 Designing the Supplier Agent

Now the design of the supplier agent is discussed. The agent-oriented design and modeling methods used are discussed. The supplier Agent is designed to

¹ downloadable from <http://www.sourceforge.net>

make the MAGNET system able to handle fully automated business-to-business interaction.

4.1 Method for high-level design

The method we chose for high-level design is Desire, by Brazier, Jonker and Treur [Brazier *et al.*, 2000]. This approach makes it easy to specify all the information types and components. The actual framework consists of more than just a design method; it includes software tools to support system design, a formal specification language, an implementation generator to automatically translate specifications into code, and verification tools for static properties of components such as consistency, correctness, completeness. We used Desire only to specify the general structure of the supplier agent, and we chose Avalon for the implementation (described later in Section 4.2), since Desire does not support Java. More details on using Desire and other tools as an agent design tool can be found in [Shehory and Sturm, 2001].

Desire specifications are based on an architecture made of components with a hierarchical relationship between them. Each component has its own input and output information types and uses its own task control knowledge, so the system is structured in a decentralized manner. The structure within the components is hidden from the “outside world”—only the interface types are defined. A component can consist of multiple sub-components if the task the component performs is too complex for one component to manage.

4.2 Method for component design

We have chosen Avalon [Loritsch, 2001] to implement the supplier agent. Using Avalon, it is straightforward to have the components of the supplier agent interact, to instantiate different instances of the components, and to reuse code. Avalon allows us to switch components on the fly, which is very useful in testing. It is also possible to configure Avalon using XML files, which specify which components and which instances have to be included.

There are seven major interfaces that can be extended in order to fit a component inside the Avalon framework: *Activity*, *Component*, *Configuration*, *Context*, *Logger*, *Parameters*, *Thread* and *Miscellany*. Each of those categories represents a unique concern area. Each component should at least implement one of these interfaces, but can also implement several of them.

The lifecycle of a component is split into three phases: *Initialization*, *Active Service* and *Destruction*. These phases occur in sequential order.

The general structure of the Avalon architecture contains the following elements: *Components*, *Component Manager*, *Component Selector* and *Component Container*. The *Components* are the cornerstones of the Avalon framework and model components as used in design methods. The *Component Manager* provides a framework for allowing components to access each other. The *Component Selector* is responsible for managing the instances of the *Components*. In order to retrieve *Components* it needs a specification of the role and a “hint” to tell

it which version to use. Finally, the *Component Container* contains the *Components* it is responsible for.

In Figure 3 we show the proposed architecture of the supplier agent using Avalon.

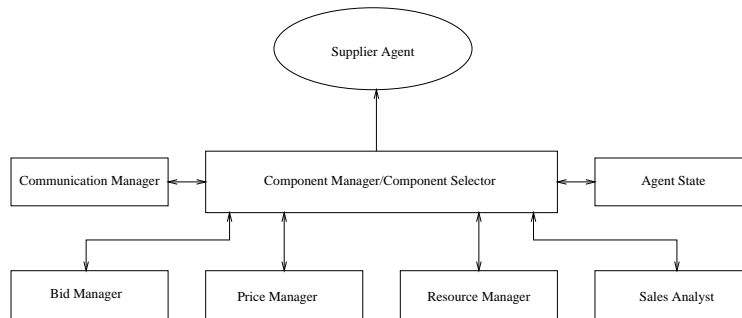


Fig. 3. Proposed architecture using Avalon

The SUPPLIERAGENT² is modeled as a *Component Container*. The SUPPLIERAGENT has no other function but to control the lifecycle of all the components. All six components are controlled by the COMPONENTMANAGER. They all get a reference to the COMPONENTMANAGER in order to be able to access the other components. If there are multiple instances of a component, the COMPONENTSELECTOR can be retrieved from the COMPONENTMANAGER and with this the right instance can be selected.

4.3 Specifying the components

We are now ready to specify the component architecture and the interfaces of the components.

Defining the component architecture The SUPPLIERAGENT is responsible for the components managed by the COMPONENTMANAGER, including COMMUNICATIONMANAGER, AGENTSTATE, BIDMANAGER, SALESANALYST, PRICEMANAGER and RESOURCEMANAGER

The task of the COMPONENTMANAGER is to manage the components owned by the SUPPLIERAGENT. It can provide its components with references to other components it manages.

The AGENTSTATE is responsible for maintaining the state of the agent and the market. This consists of keeping track of time and events coming from the

² A note to clarify the text formatting: we use *italics* to denote Avalon interfaces and ALLCAPS to refer to implemented components.

market. It also includes the saving of all information that needs to be maintained, like bids that have been submitted and incoming RFQs. Furthermore, components can subscribe to certain types of events and be notified by the AGENTSTATE when they occur.

The task of the COMMUNICATIONMANAGER is to receive information from and communicate information to the MARKET. This data includes RFQs, bids, bid awards and penalty payments.

The BIDMANAGER must decide whether to submit single or multiple bids, and whether to bid on individual tasks or block of tasks. Furthermore, it decides if the bid is to be submitted early in the bidding cycle or to wait until the last minute, and whether to bid on the entire available time windows or on some subset.

The task of the SALESANALYST is to give other components advice on whether to engage in negotiation with the customer. It keeps track of its history and relationship with various customer agents in the market. It also keeps track of sales goals for the business represented by the supplier agent. Another duty is giving advice on which task(s) to make into milestones in the bids. Milestones are tasks that require a notification be sent to the customer agent upon completion of the task.

The RESOURCEMANAGER is responsible for determining, given the current resources and the tasks, upon which tasks to bid. It also suggests upon which time windows to bid, and it is responsible for the management of the resources. This includes executing awarded tasks and handling the payments for these tasks.

The job of the PRICEMANAGER is to determine the price to bid. It can determine this price using the sales goals for these tasks, which it retrieves from the SALESANALYST. In order to determine the price, it also needs to know the cost of the resources for these tasks. This information is retrieved from the RESOURCEMANAGER.

4.4 Design using UML

The mapping between the higher-level design discussed previously and the more detailed design as presented here is completed as follows: the components become classes and the flow of information from one component to another component is accomplished through the parameters of a method call. Similarly, the information types have been modeled as classes and the information included in the information types can be accessed through method calls. When the flow of information is synchronous, the method call returns the result. When the communication is asynchronous, the information will be sent through a method call in the requesting component. A combination of both is also possible. For example, when you subscribe to a particular event you will get a confirmation through a synchronous return value. However, the event notification will be sent asynchronously.

Assume that the RFQ has already arrived at the COMMUNICATIONMANAGER and that the BIDMANAGER has already received a reference to the COMMUNI-

CATIONMANAGER and the AGENTSTATE. If aggregate transactional statistics are desired, they will be retrieved from the market.

Next, the SALESANALYST will be asked to give the BIDMANAGER advice concerning upon which tasks to focus, given its history with the customer. For this, the BIDMANAGER will have to retrieve the reference to the SALESANALYST from the COMPONENTMANAGER and, when multiple instances exist, the COMPONENTSELECTOR is also used. After the reference has been received, the BIDMANAGER will pass the RFQ to the SALESANALYST.

In our house-building example, the SALESANALYST knows that the customer that issued the RFQ has issued a RFQ before concerning the “Interior” task, and did not pay in time. The SALESANALYST decides not to include this task but to include the rest of the tasks in the advice. Another issue is to decide which milestones to include in the bids. The SALESANALYST knows that the customer finds the masonry very important, so it advises that this task be flagged as a milestone.

The reference to the RESOURCEMANAGER is retrieved in the same way as with the SALESANALYST. The BIDMANAGER will pass the task plan through to the RESOURCEMANAGER. After this has been done, the RESOURCEMANAGER checks the availability of the resources needed to complete the five tasks that are left. The RESOURCEMANAGER concludes that it is unable to provide the necessary resources for the Roofing task during the specified period. Therefore, it advises the BIDMANAGER to bid only on the other four tasks. The RESOURCEMANAGER decides to recommend the same time windows for the remaining tasks as specified in the RFQ. The resulting task plan is retrieved and will be used to determine the number of bids. In our case, the BIDMANAGER decides to issue only one bid.

Now that the bid has almost been constructed, the price needs to be calculated. For this the PRICEMANAGER is asked for advice. The reference is retrieved in the same way as described above and the task plan and the customer agent’s id are passed to the PRICEMANAGER. The PRICEMANAGER may ask for the sales goal for this customer from the SALESANALYST and the RESOURCEMANAGER will be consulted for the costs of the resources included in the task plan. The references are retrieved in the same way as with the BIDMANAGER. After the PRICEMANAGER has received the aforementioned information, it will determine the price and return it. Now the BIDMANAGER is able to make the collection of bids (one, in our case) and stores them until the time comes to issue them.

5 Current implementation of Supplier Agents

5.1 ResourceManager

Motivation. The consumption of resources by a task is analogous to the fulfillment of tasks in a task plan. The consumption of resources is considered to be a lower level of abstraction. It is therefore a necessary underpinning that allows

for the modeling of complex task fulfillment. The motivation for the RESOURCEMANAGER is to provide an abstraction for other supplier components to keep them from worrying about the details of production and consumption.

There is more to managing resources than simply modeling the consumption of consumable resources. Fixed resources, such as machines in a manufacturing cell, have a cost even when not in use. Because their use needs to be scheduled, a central part of the job of the RESOURCEMANAGER is to maintain their schedule. An additional future goal is to learn cost-minimization strategies to contribute to the SUPPLIERAGENT's profit-maximization goal and to work on methods for dynamic allocation of tasks to different resources within the supplier agent (as, for instance, in [Fatima and Wooldridge, 2001]).

Design. The design of the RESOURCEMANAGER encompasses the design of a resource production and consumption model and the design of the component.

The resource model makes two main simplifications in creating a lower level of abstraction. First, instead of considering time windows, the production and consumption of resources are quantized to discrete points in time. Second, a generic interface is presented that does not consider the explicit modeling of exotic resource attributes. Details such as price elasticity, storage, and purchasing are considered below this interface, but can still be modeled. This allows resources to simply be thought of as something the supplier agent has in time, not something the agent acquires in time.

The RESOURCEMANAGER may control many RESOURCES. Each RESOURCE acts as a wrapper to a hash table of quantized production times and corresponding amounts of reserved resources. Using this hash table, given an implementation of the productionLevel method and a confidence level, getAvailable will return the likely amount of resources available.

For each SUBTASKREQUEST in a task plan, a RESOURCE REQUIREMENT specifies the type of resource it requires and acts as an iterator to a set of evaluation times. At these evaluation times, the RESOURCE REQUIREMENT can be queried for the maximum and minimum requirements. We assume that after setting a start time, the set of evaluation times and resource requirements are deterministic.

In our current implementation, the design of the RESOURCEMANAGER is straightforward. From the interfaces mentioned above, the RESOURCEMANAGER steps through the set of consumption times using a naive strategy for resource allocation to determine satisfiability for the getTasks and getTimeWindows methods. Once a bid is made, the RESOURCEMANAGER is responsible for speculatively reserving some amount of resources. Then, once a bid award event has been received from AGENTSTATE, the RESOURCEMANAGER is responsible for the execution of those tasks. Presently, this amounts to tracking whether its resources meet the resource requirements and logging the successful or unsuccessful result.

5.2 PriceManager

Motivation. We believe that a smart PRICEMANAGER will help provide MAGNET suppliers with a competitive advantage and an incentive to join the mar-

ketplace. The lure for customers is clear: they can choose the “best deal” from those submitted. Suppliers might be hesitant to join such a market, for fear that prices will be driven too low. Our hope is that this price management technology will assure suppliers a profitable position in the market.

Design. The algorithm used by the PRICEMANAGER in this implementation endeavors to search through the space of prices for each task type and find the optimal price. That is, it looks for the price which maximizes profit as often as possible. We have developed two methods for conducting this search. One is a derivative-following (DF) technique, based on the work of Kephart [Kephart *et al.*, 2000]. The other is based on simulated annealing (SA)[Reeves, 1993].

For both methods, we track a markup percentage, so that prices reflect cost. Markups are tracked for each of the n task types the agent is interested in performing. Certainly, price tracking for sequences of tasks would provide a nice level of accuracy. However, the number of combinations would tend toward infinity as the size of the task plans increased. With long enough market exposure, the prices for individual tasks should be sufficiently shaped by their environment, so single-task schedules should provide a reasonable model.

The DF algorithm is initialized with a markup $M_i, 1 \leq i \leq n$ of 50% for each of the n task types, a maximum step size D of 50% and a price-movement direction δ of 1. When δ equals 1, the algorithm intends to raise the markup, if δ equals -1, it means to lower the percentage.

The PRICEMANAGER is called by the BIDMANAGER with a list of tasks the agent has decided to bid on (usually based on the RESOURCEMANAGER’S recommendations). A price for each task type is determined as follows. A step size is randomly chosen between zero and D . This value is multiplied by δ and added to the last bid’s markup value. Next, the cost C_i of the task in question is retrieved from the ResourceManager. The price P_i for the task is stored as $M_i * C_i + C_i$. These individual P_i ’s are totaled for all the tasks in the bid and returned to the BIDMANAGER as the bid’s price, P^t .

Once the time for awarding bids arrives, the DF PRICEMANAGER is notified by AGENTSTATE. The PRICEMANAGER takes stock of its performance and makes adjustments for the next round. A set of values $\pi_i^c, 1 \leq i \leq n$ represents the profit gained on the previous transaction. If the bid was awarded, the PRICEMANAGER calculates its profit, π_i^{new} and checks to see if $\pi_i^{new} > \pi_i^c$. If so, δ remains unchanged. However, if profit decreased or the bid was rejected, δ is switched to its opposite. π_i^c is set to the value of π_i^{new} and stored for the next bid’s comparison.

The SA PRICEMANAGER works in a somewhat similar way. Each task type’s annealing schedule is initialized with a “current” markup $M_i^c, 1 \leq i \leq n$ of 50%. The PRICEMANAGER calls the RESOURCE MANAGER and obtains from it the cost C_i of performing each task. Next a markup M_i^b is chosen for each task type. This is determined by taking a random step, less than an ever-shrinking distance D , away from the current price. (D is initialized to 0.5.) The price P_i for the task is stored as $M_i * C_i + C_i$. The prices for all the tasks are summed to give a total price, P^t , which is returned to the BIDMANAGER.

The SA PRICEMANAGER is notified by AGENTSTATE if has received an award for the work it bid on or not. The PRICEMANAGER now calculates its profit on this transaction, if any, and compares it to past performance. A set of values $\pi_i^c, 1 \leq i \leq n$ represents the best profit so far³ for each of the n types. For each task type i , M_i^b and its cost are retrieved, and the profit π_i^{new} is calculated if the bid was awarded. For rejected bids, $\pi_i^{new} = 0$. We next find the value $E = \pi_i^{new} - \pi_i^c$. If E is positive, we have made an “uphill” step, an improvement in profit. Therefore, we accept the change and P_i^c is set to the value of P_i^b . If $E < 0$, it was a “downhill” step, so we will only take it with probability $e^{E/T}$. This allows us to avoid local minima and also filters out “fluke” high profits. T is the countdown timer, so smaller and smaller steps are taken as we zero in on the profit-maximizing price.

Research is currently underway to determine the market conditions where each of these performs best.

5.3 SalesAnalyst

Motivation. The SALESANALYST can analyze customer agents and derive the most effective advising strategy, given the current customer agent. With this strategy, the profitability of the supplier agent will grow.

Design. We intend to include several features within the SALESANALYST. First of all, we would like to use data mining techniques to derive information about profiles of the customers. The information on which the data mining tool is used should be maintained within the SALESANALYST itself. This is because the information should include a history of payments and bid awards; this is considered to be private information. The profile can be used for several advising tasks: It can be used to decide on what sales goal to pass to the PRICEMANAGER. If the SALESANALYST decides that this customer should become a regular customer, it can pass a sales goal to bid a lower price. The SALESANALYST is also useful for giving advice on when not to bid, such as when the customer in question fits a “bad credit” profile. Finally, the profile of the customer can be used to decide which milestones to include in the bid.

5.4 Preliminary Experimental Results

In order to show that our design works, we ran our system using for the RFQ the example of building a house illustrated earlier in Figure 1.

Stage I The RFQ from Figure 1 was sent to each of the three suppliers.

Stage II Each supplier’s RESOURCEMANAGER chose the tasks and time windows of the bid. Since, in this case, all the suppliers bid on all the tasks, each PRICEMANAGER set a price that was near the market average price of \$14,300. The BIDMANAGER of each agent took these pieces of data and formed a single bid.

³ This is actually the profit for the last step that was accepted by the algorithm. It may not be the highest profit seen so far, if any “downhill” steps have been taken.

Stage III The bids were received and analyzed by the customer agent. It decided to accept the bid of Supplier 1.

We are testing more complex strategies for choosing which tasks to bid on and what price to bid.

6 Related Work

Markets play an essential role in the economy, and market-based architectures are a popular choice for multiple agents (see, for instance, [Chavez and Maes, 1996, Sycara and Pannu, 1998, Wellman and Wurman, 1998, Tsvetovaty *et al.*, 1997, Karacapilidis and Moraitis, 2001, Choi and Liu, 2001]). Most market architectures limit the interactions of agents to manual negotiations, direct agent-to-agent negotiation [Sandholm, 1996, Faratin *et al.*, 1997], or various types of auctions [Wurman *et al.*, 1998].

Existing architectures for multi-agent virtual markets typically rely on the agents themselves to manage the details of the interaction between them, rather than providing explicit facilities and infrastructure for managing multiple negotiation protocols. As discussed above, agents interact with each other through a Market. Our Market infrastructure provides a common vocabulary, collects statistical information that helps agents estimate costs, schedules, and risks, and acts as a trusted intermediary during the negotiation process.

Little research appears to have been done in bidding strategies for the style of auction used by MAGNET. However, Kephart, Hanson, and Greenwald have written a survey article aimed at understanding collective interactions among agents that dynamically price services or goods [Kephart *et al.*, 2000]. In this article, several useful pricing strategies for sellers are discussed. The game-theoretic computation, GT, chooses prices randomly from a distribution which is computed from buyer parameters as well as the number of sellers bidding. Like MAGNET, this pricing strategy assumes that no seller observes another seller's price before setting its own price. A second algorithm discussed is called the my-optimal, MY, or the best-response Cournot. Like the GT algorithm, MY requires perfect knowledge of the buyer population as well as the number of sellers in the Market. The third algorithm discussed is called the derivative-follower, DF. It does not require any knowledge about buyers or assumptions about the number of sellers. Rather it uses a learning technique by experimenting with increases or decreases in price. It continues to move its price in the same direction until the observed profitability level begins to fall, seeking a local maxima of profitability.

The first two pricing strategies discussed in this article could only be employed by supplier agents in MAGNET if the MAGNET server were to give out information on the number of other suppliers registered to bid on a given task. The MAGNET server shall certainly have this information, but it is not clear that this information should be given to supplier agents. The learning employed in the DF algorithm is designed to be used in situations where sales occur repeatedly. This algorithm was the inspiration for the simulated-annealing-based PriceManager.

7 Conclusion and Future Work

At this time we have a full implementation of the customer agent, but only a partial implementation of the supplier agent. In the Market, work is needed to develop mechanisms for transferring resources from the supplier to the consumer task and for tracking the monetary situation of the SUPPLIERAGENTS. In the RESOURCEMANAGER work is needed in implementing and testing the learning of cost minimization as well as resource reservation and allocation strategies. Finally, it might be interesting if the current use of task plans were to be made hierarchical. In this way, some RESOURCEMANAGERS would act as consumers until an atomic level of a task was reached.

The system is not yet mature enough to test whether the PRICEMANAGER can “learn” a good pricing strategy. Once the system is ready, we would like to see if profit margins can be improved and which features of the environment have an effect on profits. For example, we want to see the difference between markets with homogeneous supplier agents and markets with heterogeneous ones. It might also be interesting to see how the size of RFQs affect profitability. The ratio of suppliers to customers could be another interesting parameter to study.

The system does not support payments yet, so at this time it is not possible to collect the data necessary for the SALESANALYST. When these features are implemented, we would like to see whether or not we can promote loyalty among good customers and see if we can avoid high-risk tasks and ill-behaved customers.

So far, the MAGNET system has been used for several types of studies. Recent work includes experiments with performance of winner-determination algorithms [Collins *et al.*, 2002], and studies of the RFQ composition problem [Babanov *et al.*, 2002]. We are currently studying how to use an evolutionary approach to let the market develop and stabilize so that which various supplier strategies can be studied. Our longer-term goal is to support studies of supplier strategies, and studies of mixed-initiative decision making with human users in realistic market simulations.

References

- [Babanov *et al.*, 2002] Alex Babanov, John Collins, and Maria Gini. Risk and expectations in a-priori time allocation in multi-agent contracting. In *Proc. of the First Int'l Conf. on Autonomous Agents and Multi-Agent Systems*, Bologna, Italy, July 2002.
- [Brazier *et al.*, 2000] F. Brazier, C. Jonker, and J. Treur. Design of multi-agent systems. Technical report, Vrije Universiteit Amsterdam, 2000.
- [Chavez and Maes, 1996] Anthony Chavez and Pattie Maes. Kasbah: An agent marketplace for buying and selling goods. In *Proc. of the First Int'l Conf. on the Practical Application of Intelligent Agents and Multi-Agent Technology*, London, UK, April 1996.
- [Choi and Liu, 2001] Samuel P. M. Choi and Jiming Liu. A dynamic mechanism for time-constrained trading. In *Proc. of the Fifth Int'l Conf. on Autonomous Agents*, pages 568–575, 2001.

- [Collins *et al.*, 1998] John Collins, Ben Youngdahl, Scott Jamison, Bamshad Mobasher, and Maria Gini. A market architecture for multi-agent contracting. In *Proc. of the Second Int'l Conf. on Autonomous Agents*, pages 285–292, May 1998.
- [Collins *et al.*, 2002] John Collins, Maria Gini, and Bamshad Mobasher. Multi-agent negotiation using combinatorial auctions with precedence constraints. Technical Report 02-009, University of Minnesota, Department of Computer Science and Engineering, Minneapolis, Minnesota, February 2002.
- [Faratin *et al.*, 1997] Peyman Faratin, Carles Sierra, and Nick R. Jennings. Negotiation decision functions for autonomous agents. *Int. Journal of Robotics and Autonomous Systems*, 24(3-4):159–182, 1997.
- [Fatima and Wooldridge, 2001] S. Shaheen Fatima and Michael Wooldridge. Adaptive task resources allocation in multi-agent systems. In *Proc. of the Fifth Int'l Conf. on Autonomous Agents*, pages 537–544, 2001.
- [Karacapilidis and Moraitis, 2001] Nikos Karacapilidis and Pavlos Moraitis. Intelligent agents for an artificial market system. In *Proc. of the Fifth Int'l Conf. on Autonomous Agents*, pages 592–599, 2001.
- [Kephart *et al.*, 2000] Jeffrey O. Kephart, James E. Hanson, and Amy R. Greenwald. Dynamic pricing by software agents. *Computer Networks*, 32(6):731–752, 2000.
- [Loritsch, 2001] B. Loritsch. Developing with Apache Avalon. Apache Software Foundation, 2001.
- [Nisan, 1999] Noam Nisan. Bidding and allocation in combinatorial auctions. In *1999 NWU Microeconomics Workshop*, 1999.
- [OECD, 1997] OECD. Guidelines for cryptography policy. <http://www1.oecd.org/dsti/sti/it/secur/prod/crypto2.htm>, 1997.
- [Reeves, 1993] Colin R. Reeves. *Modern Heuristic Techniques for Combinatorial Problems*. John Wiley & Sons, New York, NY, 1993.
- [Sandholm, 1996] Tuomas W. Sandholm. *Negotiation Among Self-Interested Computationally Limited Agents*. PhD thesis, Department of Computer Science, University of Massachusetts at Amherst, 1996.
- [Shehory and Sturm, 2001] Onn Shehory and Arnon Sturm. Evaluation of modeling techniques for agent-based systems. In *Proc. of the Fifth Int'l Conf. on Autonomous Agents*, pages 624–631, 2001.
- [Sycara and Pannu, 1998] Katia Sycara and Anandeeep S. Pannu. The RETSINA multi-agent system: towards integrating planning, execution, and information gathering. In *Proc. of the Second Int'l Conf. on Autonomous Agents*, pages 350–351, 1998.
- [Tsvetovaty *et al.*, 1997] Maxsim Tsvetovaty, Maria Gini, Bamshad Mobasher, and Zbigniew Wieckowski. MAGMA: An agent-based virtual market for electronic commerce. *Journal of Applied Artificial Intelligence*, 11(6):501–524, 1997.
- [Wellman and Wurman, 1998] Michael P. Wellman and Peter R. Wurman. Market-aware agents for a multiagent world. *Robotics and Autonomous Systems*, 24:115–125, 1998.
- [Wurman *et al.*, 1998] Peter R. Wurman, Michael P. Wellman, and William E. Walsh. The Michigan Internet AuctionBot: A configurable auction server for human and software agents. In *Second Int'l Conf. on Autonomous Agents*, pages 301–308, May 1998.