Deliberative Stock Market Agents using Jinni and Defeasible Logic Programming

Alejandro García[1] Devender Gollapally [2] Paul Tarau[2] Guillermo Simari[1]

[1] Computer Science Department, Universidad Nacional del Sur
Avenida Alem 1253, Bahía Blanca 8000, Argentina
agarcia@cs.uns.edu.ar grs@cs.uns.edu.ar

[2] Computer Science Department, University of North Texas
Avenue B at Mulberry, Denton, Texas 76203, U.S.A.
gollapal@silo.csci.unt.edu tarau@silo.csci.unt.edu

Abstract. Working with stock markets requires constant monitoring of the stock information which keeps changing continously, and the ability to take decisions instantaneously based on certain rules as the changes occur. In this paper, a framework for implementing a Deliberative Multi-Agent System is developed. This system can be used as a proactive tool for expressing and putting to work high level stock trading strategies. In this framework, agents are able to monitor and extract stock market information via the World Wide Web and, using the domain knowledge provided in the form of defeasible rules, can reason in order to achieve the established goals.
The overall system is integrated using Jinni which provides a platform for building intelligent autonomous agents. The agents have a Reasoning Module, based on Defeasible Logic Programming, capable of formulating arguments and counterarguments in order to decide whether or not to perform an action. Arguments and counterarguments are compared and a thorough dialectical analysis is performed. From the outcome of this analysis agents are able to decide which action to perform based on the support of warranted arguments, as real world agents would do.

1 Introduction

In this paper, a framework for implementing a Deliberative Multi-Agent System is developed. This framework can be used as a proactive tool for expressing and putting to work high level stock trading strategies. The agents described here are able to monitor the stock market, extract information, and using the domain knowledge provided in the form of defeasible rules can perform defeasible reasoning in order to achieve the established goals.

This project combines Jinni [18] (Java INference engine and Networked Interactor) and an implementation of Defeasible Logic Programming (DeLP) [5, 6]. Jinni is used as a platform for building intelligent autonomous agents, and DeLP provides the agents with the capability of reasoning using defeasible rules in a dynamically changing domain.

Working with the stock market requires constant monitoring of fluctuating data such as respective indexes, beta factors, value line analysis, large amounts of financial information, up-to-date stock quotes, as well as the ability to recall historical data. An agent working in such an environment usually keeps track of the fluctuations of a particular portfolio, and then, based on the stock information has to decide when to perform some action, like selling or buying stocks. For doing so, domain knowledge in the form of defeasible rules is normally used, and arguments and counterarguments are formulated to take decisions.

Agents as described in this paper can perform all the above-mentioned tasks with little effort on the user's part. They can monitor the world market via Internet, collect up-to-date stock values, gather the risk factors, monitor market indexes, and collect information from various specialized web pages in a manner that closely resembles a human agent. The stock information gathering is based on a Stock Retrieval Module implemented in Java as a set of built-in Jinni functions.

In this framework several agents can be programmed for monitoring the stock market and performing actions based on the retrieved information. The agents have a Reasoning Module, based on DeLP, capable of formulating arguments and counterarguments in order to decide whether to perform an action or not. For instance, an agent can be programmed to alert the user when it is the right moment for buying some stock.

In DeLP, knowledge can be expressed in the form of defeasible rules. That is, rules that provide a weak link liable to exceptions between the antecedent and the consequent. Defeasible rules are used to represent tentative information that may be used if nothing could be posed against it, e. g., usually buy a ticker if the price is good. Since DeLP is based on a defeasible argumentation formalism, the reasoning module of the agents will use defeasible rules and the information retrieved from the stock market for building arguments and counterarguments for and against performing some action. In DeLP, arguments and counterarguments are compared using a particular criterion, and an exhaustive dialectical analysis is performed (see Section 5). In that way, agents can decide which action to perform based on warranted arguments, as real world agents do.

In order to obtain specific information from the stock market, the agents will consult with other agents and a remote database to gather historical or current data. This database will be updated by agents dedicated to that chore (see section 4).

All the different components: the DeLP based agents, the Stock Retrieval agents, and the database updater, are integrated together as one system using Jinni as the common platform. Jinni has the advantage of allowing Java and Prolog programs to run in the same environment and provides primitives for programming mobile agents.

This paper is organized as follows. In section 2, an outline of the system is introduced. Section 4 describes the Stock Retrieval agents, and the Stock Information Data Base. Section 5 presents a brief description of Defeasible Logic

Programming, and later in section 6 agents that use DeLP are introduced. Finally, related approaches and future work are discussed.

2 Overview of the framework

As stated before, constant monitoring of the stock market information which keeps changing continously and the ability to instantaneously take decisions based on certain rules as the changes occur are important aspects of working with stocks. In this framework we have used Jinni as the platform for building a multi-agent architecture where agents have different roles (buy or sell stocks, Web information extraction) and cooperate asynchronously through blackboards. Defeasible Logic Programming provides defeasible reasoning capabilities.

Jinni [18] is a lightweight, multi-threaded, logic programming language, intended to be used as a flexible scripting tool for gluing together knowledge processing components and Java objects in distributed applications. Jinni is also a convenient development platform for distributed AI, and in particular, for building intelligent autonomous agent applications. In Jinni, Prolog terms can be stored and retrieved by local or remote agents using a blackboard.

Defeasible Logic Programming [6] (DeLP) is an extension of Logic Programming capturing common-sense reasoning features that are difficult to express in traditional Logic Programming. DeLP can manage defeasible reasoning, allowing the representation of defeasible and non-defeasible rules. In DeLP, answers to queries are supported by arguments obtained using program rules. Arguments may also be defeated by other arguments called counterarguments. Informally, a query q succeeds if a supporting argument for q cannot be defeated and in that manner becomes a warranted conclusion. In order to establish whether $\mathcal{A}$ is a non-defeated argument, counter-arguments that could be defeaters for $\mathcal{A}$ are considered, i. e., counter-arguments that for some criterion, are preferred to $\mathcal{A}$. Since counter-arguments are arguments, there may exist defeaters for them, and so on. In DeLP an exhaustive dialectical analysis is performed building a dialectical tree. Thus, all arguments for and against a conclusion are taken in consideration. In Section 5 we describe DeLP in more detail.

In this framework, several agents can be programmed for monitoring the stock market and for performing some actions based on the retrieved information. The overall system is integrated under Jinni. Figure 1 shows an outline of the architecture. The system consists of information extraction agents specialized in gathering stock market data from particular web pages, and deliberative agents consulting the information extraction agents and analyzing the retrieved information in order to perform actions. The agents can be running in the same or different machines, and the comunication is performed using a Jinni blackboard. The information extraction agents can retrieve information on their own or they can retrieve particular information requested by other agents. Each information extraction agent consists of

1. a Stock Retrieval Module (SRM): a java object that extracts automatically stock values and advises from the Internet (explained in Section 4), and

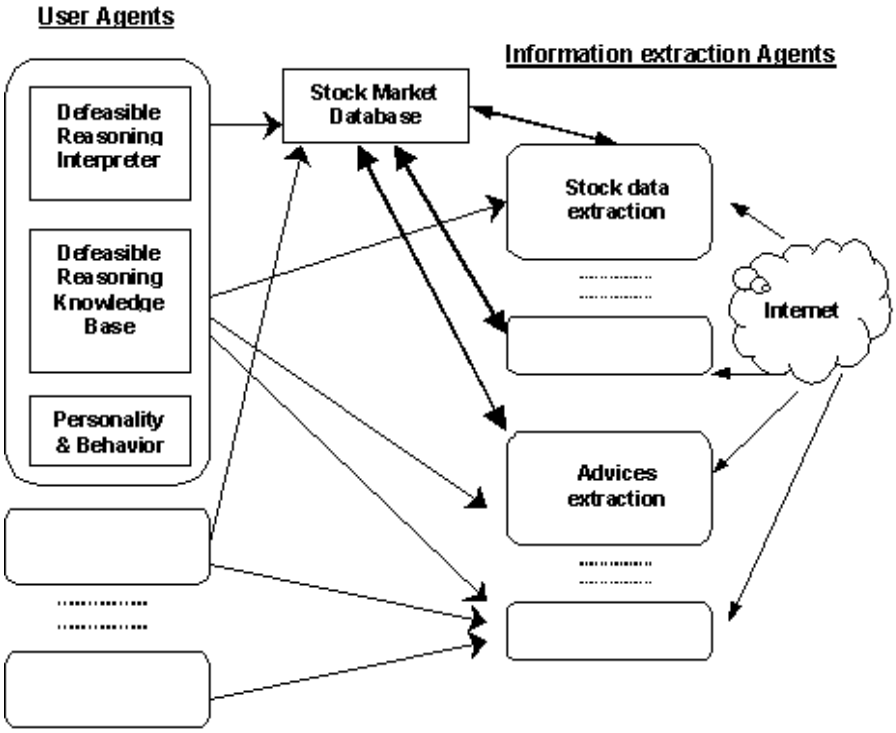

Fig. 1. System outline

2. a module implemented in Jinni that updates the Stock Market Data Base calling SRM with a particular Stock Information (see section 4).

The Stock Market Data Base contains current and historical stock information retrieved by the information extraction agents. The information that the agents retrieve consists of stock quotes, and advises from the Internet. All this information is stored in the form of Prolog terms in Jinni's blackboard, thus, accessible by any user agent. Agents can also use the information of the Stock Market Data Base for generating historical data.

The user agents are deliberative entities that can perform actions based on established goals and the information present in the Stock Market Data Base. Each agent as described in detail in Section 6, consists of:

1. Domain Knowledge, expressed as a DeLP program,
2. a Defeasible Reasoning Module implemented with a DeLP interpreter,
3. the specification of goals and actions to be performed.

Goals will be handled by the agent's Reasoning Module using specific domain information considered as part of each agent's knowledge, and information

provided by information extraction agents. Agents that work with a particular portfolio can consult different information extraction agents for obtaining data. Obtaining information of differents sources leads to possible contradictions in the view of the world as described by the retrieved data. A plausibility criterion for deciding between contradictions among different sources is part of the future work on this framework.

3 Jinni

The system developed here is based on Jinni (Java INference engine and Networked Interactor). In this section we give a brief overview of Jinni and we refer the interested reader to [19] for details. Jinni is a lightweight, multi-threaded, logic programming language, intended to be used as a flexible scripting tool for gluing together knowledge processing components and Java objects in distributed applications.

Jinni is a convenient development platform for distributed AI, and in particular, for building intelligent autonomous agent applications. It has been developed in Java, which makes it platform independent. Jinni allows us to include and use Java objects and it also includes lightweight Prolog interpreters. Java objects can be called from a Jinni interpreter like normal predicates. Thus, Jinni combines the advantages of both Object Oriented Programming and logic programming. For example the predicates nasdaq/3 and cnnadvisor/2 (see Section 4) are actually implemented in Java and can be used as a Jinni built-in.

Jinni also includes high level networking operations allowing code mobility and remote execution. This allows agents to access remote Jinni servers and extract or update information on the server, and also execute code based on the server's knowledge base.

Jinni agents can access local and remote blackboards and assert or consult Prolog terms from there using out/1, in/1, cout/1 and cin/1. On the one hand, the predicates out(Term) and cout/1 put a term on a local or remote blackboard and resumes execution of threads waiting for a matching term. cout/1 does nothing if a matching term is already on the blackboard. On the other hand, in(Term) waits until a pattern unifiable with Term is put on the (remote or local) blackboard by an out/1 or cout/1 operation. cin(X) removes and returns a term currently on the (remote or local) blackboard matching X and fails if no such term exists.

In this framework the blackboard is used for implementing the communication among agents, and for maintaining a knowledge base. It can also be used to co-ordinate agents. Various information provider agents can update the Jinni server's knowledge base, and then other agents can retrieve information from there.

In most agent architectures, communication has been implemented using message queues. However, Jinni provides a more general way of communication and coordination among agents using blackboards. In this framework, a blackboard is the central hub for all the communication among agents. For instance, when the user agent starts running and consults the blackboard for the latest

stock ticker values, it will wait as a suspended thread until some agent fills in the required information.

4 Retrieving Stock Information

As stated before, the information extraction agents retrieve stock data and advice for selling or buying stocks from particular web pages. The information extraction agents can retrieve information on their own or they can retrieve particular information asked by other agents.

Once the information is retrieved the agents update the Stock Market Data Base on the Jinni's blackboard. The agents work continously, connecting periodically to Internet in order to keep the Stock Data Base updated. Ensuring in this way that the prices returned are always the most recent.

For retrieving particular values for a given stock, Jinni was extended with a new built-in called nasdaq/3. The built-in was implemented in java and can be called as any other regular Prolog predicate. When nasdaq/3 is called, it opens a socket to a particular web site and downloads the web page. This page is then parsed and the required information is extracted.

The built-in nasdaq/3 has 3 arguments: nasdaq(Ticker, TypeOfData, Value).

- Ticker: is the name of the stock symbol being consulted, for example "yhoo" (Yahoo), "ti" (Texas Instrument), "amzn" (Amazon) .
- TypeOfData: This parameter is the type of information consulted, for example, today's low value, today's high, the shared volume, last sale value.
- Value: Is the result of the type of information consulted for the given Ticker.

Here are some examples: nasdaq(amzn,todayslow,X); nasdaq(csco,todayshigh,X); nasdaq(txn,previousclose,X) nasdaq(yhoo,netchange,X); nasdaq(ti,lastsale,X).

A similar implementation was done for retrieving advice for particular stocks. There is a java object registered in Jinni as a built-in, called cnnadvisor/2. This object is responsible for extracting advice from a particular web page for buying or selling a stock. The syntax for the predicate is cnnadvisor(Ticker, Advice), where Advice could be: "strong buy", "buy", "sell", "strong sell".

Information extraction agents are programmed in Jinni using the new built-ins nasdaq/3 and cnnadvisor/2. One agent could be programmed for looping forever retrieving information only for "amazon" stocks. For doing so, it will use nasdaq(amzn,T, Value), for different values for T: todayslow, todayshigh, previousclose, lastsale,etc. Other agent could be programmed for monitoring continously the same type of data for different stocks. For example, for monitoring the last value of a set of stocks, the agent will use nasdaq(S,lastsale, Value), for the different stock symbols in S: yhoo, ti, csco, amzn, etc. Similarly, agents could be continuously retrieving advice for selling or buying particular stocks.

Agents use the cin/1 and cout/1 Jinni's predicates for updating the Data Base in the remote blackboard. The information is stored in the blackboard as Prolog facts, and can be consulted by the user agents through their Reasoning Module.

An information extraction agent A could also be inactive, waiting for some query. If A is then activated by an other agent B for obtaining the value of the last sale for Yahoo stocks, the agent A will use the built-in nasdaq(yhoo,lastsale, X) obtaining an actual value v for X and then it will generate the fact todayslow(amzn,v) which will be stored in the blackboard. Thus, communicating with the agent B which it is waiting for the information.

5 Defeasible Logic Programming

We include here a brief description of DeLP, and refer the interested reader to [7] for details. The DeLP language is defined in terms of two disjoint sets of rules: a set of strict rules for representing strict (sound) knowledge (denoted $L_0 \leftarrow L_1, \ldots, L_n$), and a set of defeasible rules for representing tentative information (denoted $L_0 -\!\prec L_1, \ldots, L_n$). In both types of rules, the head L_0 and the elements of the body $L_1, \ldots, L_n$, are literals, i. e., atoms or negated atoms using strong negation ($\sim$). When the body is empty, a strict rule is called a fact, and a defeasible rule a presumption. Defeasible Rules add a new representational capability for expressing a weaker link between the head and the body in a rule. A defeasible rule "$Head -\!\prec Body$" is understood as expressing that "reasons to believe in the antecedent, $Body$, provide reasons to believe in the consequent, $Head$" [17]. A DeLP program is a finite set of defeasible and strict rules.

Example 1. Here follows a DeLP program:

$buy(T) -\!\prec good\_price(T)$
$\sim buy(T) -\!\prec good\_price(T), price\_decreasing(T)$
$price\_decreasing(T) -\!\prec lastsale(T,P), previousclose(T,Pre), P < Pre$
$good\_price(ti) -\!\prec lastsale(T,P), P < 100$

In DeLP an argument $\mathcal{A}$ for a literal q is a minimal and non-contradictory derivation for q (see [7] for details). An argument $\mathcal{A}_1$ is a subargument of $\mathcal{A}$ if $\mathcal{A}_1 \subseteq \mathcal{A}$. An argument $\mathcal{B}$ is a counter-argument for an argument $\mathcal{A}$, if $\mathcal{B}$ attacks (is in contradiction with) some internal subargument of $\mathcal{A}$ called the disagreement subargument.

Example 2. Consider that a server data base contains the following information: "$lastsale(ti, 70)$" and "$previousclose(ti, 85)$". Using this information and the rules of Example 1, there is an argument $\mathcal{A}$ for "$buy(ti)$"

$$\mathcal{A} = \left\{ \begin{array}{l} buy(ti) -\!\prec good\_price(ti) \\ good\_price(ti) -\!\prec lastsale(ti, 70), 70 < 100 \end{array} \right\}$$

and also an argument $\mathcal{B}$ for "$\sim buy(ti)$".

$$\mathcal{B} = \left\{ \begin{array}{l} \sim buy(ti) -\!\prec good\_price(ti), price\_decreasing(ti) \\ price\_decreasing(ti) -\!\prec lastsale(ti, 70), previousclose(ti, 85), 70 < 85 \end{array} \right\}$$

In this situation, $\mathcal{B}$ is a counter-argument for $\mathcal{A}$, and viceversa.

Arguments and counter-arguments are compared using a particular criterion called specificity which prefers arguments based on more information, and arguments with shorter derivations. Also, arguments that are based on facts are preferred over arguments based on presumptions. A counter-argument $\mathcal{B}$ is a proper defeater for $\mathcal{A}$ when $\mathcal{B}$ is better than a disagreement subargument of $\mathcal{A}$. A counter-argument $\mathcal{B}$ is a blocking defeater for $\mathcal{A}$ when $\mathcal{B}$ and the disagreement subargument are unrelated with respect to the comparison criterion. Following Example 2, $\mathcal{B}$ is more specific than $\mathcal{A}$, and therefore a proper defeater for it.

In DeLP a literal q will be warranted if there is a supporting argument $\mathcal{A}$ that could not be defeated. In order to establish that $\mathcal{A}$ is a non-defeated argument, all possible defeaters for $\mathcal{A}$ will be considered. Since defeaters are arguments, defeaters for those defeaters will in turn be considered, and so on. Hence, an argument $\mathcal{A}$ will be defeated if there exists at least one defeater $\mathcal{D}$ for $\mathcal{A}$, and $\mathcal{D}$ is not defeated.

Observe that there could be more than one defeater for a given argument, so this recursive specification leads to the consideration of a tree structure. The root of the tree corresponds to the argument $\mathcal{A}$ and every inner node represents a defeater (proper or blocking), of its parent. Leaves in this tree correspond to non-defeated arguments. This structure is called a dialectical tree and was first introduced in [16]. Thus, in DeLP an exhaustive analysis of arguments and their defeaters is performed, considering all the possible reasons for and against the literal under discussion.

Example 3. Consider the following DeLP program:

$sell\_stock(T) \prec advice(T, sell)$
$\sim sell\_stock(T) \prec advice(T, sell), speculate(T)$
$speculate(T) \prec negative\_profit(T)$
$\sim speculate(T) \prec negative\_profit(T), too\_risky(T)$
$too\_risky(T) \prec market(down)$
$too\_risky(T) \prec breaking(T)$
$negative\_profit(T) \leftarrow pricepaid(T, P), lastsale(T, L), L < P$
$pricepaid(msft, 101)$

and suppose the server data base contains the following information
"$advice(msft, sell) \prec true$" "$lastsale(msft, 70)$" and "$market(down)$"
Using this information, there is an argument $\mathcal{A}$ for $sell\_stock(msft)$ based on the advice, that it is defeated by the argument $\mathcal{B}$ for $\sim sell\_stock(msft)$, however, this last argument is in turn defeated by an argument $\mathcal{C}$ for $\sim speculate(msft)$. No argument can defeat $\mathcal{C}$, so argument $\mathcal{A}$ is reinstated because there is no undefeated defeater for it.

In order to ensure that the dialectical process ends, DeLP detects and avoids cycles in the dialectical process (see [6, 16]). In DeLP there are four possible answers for a query "h":

– yes, if exists an argument $\mathcal{A}$ for h that is a warrant (has no undefeated defeater).

– no, if for each possible argument $\mathcal{A}$ for h, there exists at least one proper defeater for $\mathcal{A}$ that remains undefeated.
– undecided, if h is not warranted, and there exists an argument $\mathcal{A}$ for h, such that there are no proper defeaters undefeated, but there exists at least one blocking defeater that remains undefeated.
– unknown, if there exists no argument for h.

6 Agents with Defeasible Reasoning

An agent is specified given some goals and some actions to be performed when these goals are achieved. For instance, the agent's goal could be "buy stock of a company C" and the associated action to this goal could be "let me know when the goal can be achieved". These goals are used by the agent as queries to the reasoning module (DeLP interpreter).

Agents are implemented as a Prolog program (see Figure 2). However, an appropriate user interface provides a way of specifying the agent behavior without knowing how to program in Prolog. The strict and defeasible rules are also specified using a graphical interface.

```
try(Goal):- set˙host('csci.unt.edu'),  % select one of possible hosts
            repeat,   % loop and ask every 5 seconds until answer is yes
            sleep(5),
            answer(Goal,Argument,Answer),  % call to DLP Interpreter
            Answer = yes,
           write('The goal: '), write(Goal), write(' can be achieved '),nl,
           write('because: '),write(Argument),nl.
```

Fig. 2. Agent's implementation in Bin Prolog

As showed in Figure 2, the DeLP interpreter is invoked using the predicate answer/3. Given a Goal, the predicate answer/3 returns an Argument and the Answer for that query. As stated before, in DeLP there are four possible answer for a query: yes, no, undecided, and unknown. The agent will take some action, depending on the answer. In the case of the agent described on Figure 2, it will inform the user when the answer is yes. If the answer does not match, then the agent will try periodically until it obtains the correct answer. A timer could also be set in order to prevent the agent running forever.
In order to answer a query, the DeLP interpreter uses:

1. the strict and defeasible rules of the agent's domain knowledge, and
2. the current information in the server Data Base.

Example 4. Consider that agent of Figure 2 is called with the goal $sell\_stock(msft)$, and has the set of rules of Example 3 as its domain knowledge:

$sell\_stock(T) \prec advice(T, sell)$
$\sim sell\_stock(T) \prec advice(T, sell), speculate(T)$
$speculate(T) \prec negative\_profit(T)$
$\sim speculate(T) \prec negative\_profit(T), too\_risky(T)$
$too\_risky(T) \prec market(down)$
$too\_risky(T) \prec breaking(T)$
$negative\_profit(T) \leftarrow pricepaid(T, P), lastsale(T, L), L < P$
$pricepaid(msft, 101)$

Suppose the server data base contains the following information "$advice(msft, sell) \prec true$" "$lastsale(msft, 70)$" and "$market(up)$.

With all this information, if the agent specified in Figure 2 submit to the DeLP interpreter the query "$sell\_stock(msft)$", the answer will be no, because the argument $\mathcal{A}$ for $sell\_stock(msft)$

$$\mathcal{A} = \left\{ \begin{array}{l} sell\_stock(msft) \prec advice(msft, sell) \\ advice(msft, sell) \prec true \end{array} \right\}$$

has the following defeater $\mathcal{B}$ for $\sim sell\_stock(msft)$ and $\mathcal{B}$ is undefeated.

$$\left\{ \begin{array}{l} \sim sell\_stock(msft) \prec advice(msft, sell), speculate(msft) \\ speculate(msft) \prec negative\_profit(msft) \\ negative\_profit(msft) \leftarrow pricepaid(msft, 101), lastsale(msft, 70), 70 < 101 \\ pricepaid(msft, 101) \\ advice(msft, sell) \prec true \\ lastsale(msft, 70) \end{array} \right\}$$

Since the answer is no, the agent will continue trying periodically waiting for some change in the stock information. Suppose that the data base is updated and the status of the market changes to $market(down)$. With this information an argument $\mathcal{C}$ for $\sim speculate(msft)$ can be build. The argument $\mathcal{C}$ becomes a defeater for $\mathcal{B}$ (attacking an inner point).

$$\left\{ \begin{array}{l} \sim speculate(msft) \prec negative\_profit(msft), too\_risky(msft) \\ too\_risky(msft) \prec market(down) \\ negative\_profit(msft) \leftarrow pricepaid(msft, 101), lastsale(msft, 70), 70 < 101 \\ pricepaid(msft, 101) \\ market(down) \end{array} \right\}$$

Now, argument $\mathcal{A}$ is defeated by $\mathcal{B}$, and $\mathcal{B}$ is defeated by $\mathcal{C}$. Since there is no defeater for $\mathcal{C}$, $\mathcal{B}$ is defeated, and $\mathcal{A}$ is reinstated. Thus the answer for the query "$sell\_stock(msft)$" will be yes, and the agent will inform the user that "The goal $sell\_stock(msft)$ can be achieved".

It is interesting to note that the agent could also perform an action based on a different answer, like undecided. This will allow to specify different personalities for the agent.

Agent personalities

There are four possible answers for a query (yes, no, undecided, unknown) and the argument that supports the answer could be based on facts or presumptions. Using this information, it is possible to specify different personalities for agents.

For instance, a "safe" agent could be one that performs action only when the answers for queries are yes, as agent of Figure 2. A more "risk-taker" or "aggressive" agent could be specified if actions are performed when the answer for the goal is yes or undecided. The following table shows some possible personalities.

Personality	when answer to goal is:	and the argument is based on
very safe	yes	facts
safe	yes	presumptions
aggressive	yes or undecided	facts
risk-taker	yes or undecided	presumptions
brave	yes or undecided or unknown	–

7 Related and Future Work

In [11] and [20] an architecture for an agent-based virtual market is proposed. The system includes all elements required for simulating a real market, including a communication infrastructure, mechanisms for storage and transfer of goods, banking and monetary transactions, and economic mechanisms for direct or brokered producer-consumer transactions. In our case, all the communication infrastructure is provided by Jinni. In contrast with [11], we were interested first in developing deliberative autonomous agents that can extract information directly from web pages, reason and take decision based on this information. As a future work we will considerate banking transaction issues.

Various techniques for fast and efficient data mining, which includes machine learning, are discussed in [9]. As described here, we have primitives to gather information from Internet, and the "updating agents" also produce new historical information useful for the user agents. We consider this as a future extension using the techniques introduced in [9].

Mikler and Mayes in [12] focus on a distributed approach to data mining stock values form the Internet, which reduce the bottle neck in extracting data. In our framework different agents may extract different types of data, reducing the overall time in extraction. In [4] Ying presents the first steps towards creating automated bargaining intelligent assistants that can reason about the relative supply and demand for goods and services and negotiate to reach a good deal. Parkes in [13] presents a multi-agent model for the multiperiod portfolio selection problem. Individual agents each receive a share of initial wealth, and follow an investment strategy to adjust their portfolio as they observe movements of the market over time. They also suggested that a cooperative multiagent system, with a simple communication mechanism of explicit hint exchange between agents, achieves a further increase in performance.

There are several related work in argumentation and defeasible logic [1, 10, 15, 21, 22, 2]. However, as far as we know, using argumentation for stock market agents was not considered in previous approachs. The interested reader is referred to the following surveys in defeasible argumentation: [14, 3].

We are working now in the development of a more robust, efficient and fast system for data extraction based on distribute data mining. The framework described here can easily be extended to provide the user with a global market place where agents can bargain. The agents could sell or buy products (or services) on behalf of their owners and also provide business to business services.

8 Conclusion

The framework developed here allows to define agents for expressing and putting to work high level stock trading strategies. Agents with different strategies and also different personalities can be implemented easily. The paper shows that an interesting synergy exists between network enabled deliberative agent programs and Defeasible Logic Programming used as a dynamically adjustive reasoning mechanism. Personality choices can be parameterized to express various trading strategies.

Acknowledgments

Special thanks to Dr. Armin R. Mikler, Yu Zhang, and Jeff Barrett, and to the anonymous referees for their suggestions.

References

1. Grigoris Antoniou, Michel J. Maher, and David Billington. Defeasible logic versus logic programming without negation as failure. Journal of Logic Programming, 41(1):45–57, 2000.
2. A. Bondarenko, P.M. Dung, R.A. Kowalski, and F. Toni. An abstract, argumentation-theoretic approach to default reasoning. Artificial Intelligence, 93:63–101, 1997.
3. C. I. Chesñevar, A. Maguitman, and R.P.Loui. Logical models of arguments. submitted to ACM Computing Surveys, 1998.
4. Ying Sun Daniel. Automated bargaining agents. In S. Weld Department of Computer Science and Engineering, FR-35 University of Washington Seattle, WA 98195, January 1995.
5. Alejandro J. García. Defeasible logic programming: Definition and implementation. Master's thesis, Dep. de Ciencias de la Computación, Universidad Nacional del Sur, Bahía Blanca, Argentina, July 1997.
6. Alejandro J. García and Guillermo R. Simari. Defeasible logic programming. Technical report, Computer Science Department, Universidad Nacional del Sur, October 1998. Technical Report GIIA-1998-20.

7. Alejandro J. García and Guillermo R. Simari. Sources of parallelism in defeasible logic programming. In Proceedings of the IV Congreso Argentino en Ciencias de la Computación, October 1998.
8. Alejandro J. García, Guillermo R. Simari, and Carlos I. Chesñevar. An argumentative framework for reasoning with inconsistent and incomplete information. In Workshop on Practical Reasoning and Rationality. 13th biennial European Conference on Artificial Intelligence (ECAI-98), August 1998.
9. Rishi Nayar Jonny S. K. Wong and Armin R. Mikler. A framework for a world wide web-based data mining system. Journal of Network and Computer Applications, pages 163–185, 1998.
10. Robert A. Kowalski and Francesca Toni. Abstract argumentation. Artificial Intelligence and Law, 4(3-4):275–296, 1996.
11. B. Mobasher M. Tsvetovatyy, M. Gini and Z. Wieckowski. Magma: An agent-based virtual market for electronic commerce. Journal of Applied Artificial Intelligence, 6, 1997.
12. Armin R. Mikler and John T. Mayes. Distributed data mining - an application for wos. In Distributed Computing on the Web (DCW '99). Society of Computer Science(Gesellschaft fur informatik), June 1999.
13. David C. Parkes. Multiagent cooperative search for portfolio selection. Computer and Information Science Department University of Pennsylvania Philadelphia.
14. H. Prakken and G. Vreeswijk. Logical systems for defeasible argumentation (to appear). In D.Gabbay, editor, Handbook of Philosophical Logic, 2nd ed. Kluwer Academic Pub.
15. Henry Prakken. Logical Tools for Modelling Legal Argument. A Study of Defeasible Reasoning in Law. Kluwer Law and Philosophy Library, 1997.
16. Guillermo R. Simari, Carlos I. Chesñevar, and Alejandro J. García. The role of dialectics in defeasible argumentation. In XIV International Conference of the Chilenean Computer Science Society, November 1994.
17. Guillermo R. Simari and Ronald P. Loui. A Mathematical Treatment of Defeasible Reasoning and its Implementation. Artificial Intelligence, 53:125–157, 1992.
18. Paul Tarau. Inference and Computation Mobility with Jinni. In K.R. Apt, V.W. Marek, and M. Truszczynski, editors, The Logic Programming Paradigm: a 25 Year Perspective, pages 33–48. Springer, 1999. ISBN 3-540-65463-1.
19. Paul Tarau. Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. In Proceedings of The Fourth International Conference on The Practical Application of Intelligent Agents and Multi-Agents, pages 109–123, London, U.K., 1999.
20. M. Tsvetovatyy and M. Gini. Toward a virtual marketplace: Architectures and strategies. In PAAM96. PAAM, 1996.
21. Bart Verheij. Argue! an implemented system for computer-mediated argumentation. In Proc. of the 10th Netherlands/Belgium Conference on Artificial Intelligence, pages 57–66. CWI, Amsterdam, 1998.
22. Gerard A.W. Vreeswijk. Abstract argumentation systems. Artificial Intelligence, 90:225–279, 1997.