DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY
Office for Scholarly Communication

Specifying and Monitoring Economic Environments Using Rights and Obligations

Citation
Michael, Loizos, David C. Parkes, and Avi Pfeffer. 2010. Specifying and monitoring economic environments using rights and obligations. Autonomous Agents and Multi-Agent Systems 20(2): 158-197.

Published Version
doi:10.1007/s10458-009-9089-6

Permanent link
http://nrs.harvard.edu/urn-3:HUL.InstRepos:3967324

Terms of Use
This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Open Access Policy Articles, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#OAP

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story .

Accessibility

Specifying and Monitoring Economic Environments using Rights and Obligations*

Loizos Michael† David C. Parkes‡ Avi Pfeffer§

March 22, 2009

Keywords: market semantics; rights; obligations; electronic transactions

Abstract

We provide a formal scripting language to capture the semantics of economic environments. The language is based on a set of well-defined design principles and makes explicit an agent's rights, as derived from property, and an agent's obligations, as derived from restrictions placed on its actions either voluntarily or as a consequence of other actions. Coupled with the language is a run-time system that is able to monitor and enforce rights and obligations in an agent-mediated economic environment. The framework allows an agent to formally express guarantees (obligations) in relation to its actions, and the run-time system automatically checks that these obligations are met and verifies that an agent has appropriate rights before executing an action.

Rights and obligations are viewed as *first-class* goods that can be transferred from one agent to another. This treatment makes it easy to define natural and expressive recursive statements, so that, for instance, one may have rights or obligations in selling or trading some other right or obligation. We define *fundamental axioms* about well-functioning markets in terms of rights and obligations, and delineate the difference between *ownership* and *possession*, arguably two of the most important notions in economic markets. The framework provides a rich set of action-related constructs for modeling conditional and non-deterministic effects, and introduces the use of transactions to safely bundle actions, including the issuing of rights and taking on of obligations. By way of example, we show that our language can represent a variety of economic mechanisms, ranging from simple two-agent single-good exchanges to complicated combinatorial auctions. The framework, which is fully implemented, can be used to formalize the semantics of markets; as a platform for prototyping, testing and evaluating agent-mediated markets; and also provide a basis for deploying an electronic market.

*A preliminary version of this work appeared in the *Proceedings of Sixth International Workshop on Agent Mediated Electronic Commerce (AMEC'04)*, P. Faratin and J. A. Rodriguez-Aguilar (Eds.), Vol. 3435, pp. 188–201, Lecture Notes in Artificial Intelligence, Springer-Verlag, 2005. This paper is a significantly extended version, with discussions on the types of obligations and the possible applications of this work, and with numerous new illustrative examples, including detailed representations of typical economic environments, along with specifics about the monitoring system, the scripting language, objects, states, transactions and effects that were omitted from the earlier version.

†Department of Computer Science, University of Cyprus, CY-1678 Nicosia, Cyprus. Email `loizosm@cs.ucy.ac.cy`. This work was completed while the author was at the School of Engineering and Applied Sciences, Harvard University, Cambridge, MA 02138, U.S.A.

‡School of Engineering and Applied Sciences, Harvard University, Cambridge, MA 02138, U.S.A. Email `parkes@eecs.harvard.edu`

§School of Engineering and Applied Sciences, Harvard University, Cambridge, MA 02138, U.S.A. Email `avi@eecs.harvard.edu`

1 Introduction

Many authors have written about a future of agent-mediated electronic commerce, in which agents engage in commerce on behalf of individuals and businesses [23, 27, 30, 36, 46, 48, 50, 51, 63]. Part of the challenge of agent-mediated commerce is one of providing trust and reliability, so that agents act, and can be seen to act, in the interests of the parties that they represent, be they individuals, firms, or other groups of people. This motivates our formal scripting language for describing economic markets that is: *(i)* natural and easy to understand, for humans to be able to participate, *(ii)* formal and unambiguous, for artificial agents to be able to participate, and *(iii)* amenable to automatic monitoring.

The need for a formal method to describe markets in a computer-compliant yet human-friendly way naturally arises in a variety of contexts. Most prevalent is that of online transactions between agents, including both humans and artificial bidding agents. Our framework can be used to deploy an actual market that is open to agents joining and participating in economic transactions. An equally important context is the need for a platform for testing new agent designs, simulating new mechanism designs, and evaluating their properties. Our framework can provide an important tool for designers and prospective market participants alike. Indeed, trading agent competitions with simulated markets have provided new impetus to the problem of bidding agent design [52, 61]. We enable a well-specified and functional "sand box" for the development of such market environments, including also for the monitoring and validation of particular market mechanisms and agent behaviors.

The scripting language we propose captures the essential semantics, namely *rights* and *obligations*, of economic environments. Rights enable agents to obtain utility by taking actions on goods that they own or possess, while obligations allow them to engage in safe transactions and make credible commitments to rules of encounter [28, 46]. We adopt rights and obligations as *first-class* goods, and derive fundamental market axioms. The axioms delineate the difference between *ownership* and *possession* of goods, and allow agents to manipulate goods as a result of derived rights. Additional axioms provide a natural means to implement *private information*, a central notion in economic markets. The provisions of axioms apply recursively on rights and obligations themselves. These axioms are enforced through a monitoring system that we couple with our formal scripting language. Given a description of an economic environment, the monitoring system implements the domain in a prescribed way, thus giving precise semantics to the scripting language.

Agents can interact with the monitoring system and affect, through their actions, the state of the economic environment. For example, an agent can initiate a new economic mechanism by specifying obligations on its behavior (e.g., "I will sell to the highest bidder.") and granting rights to participants (e.g., "All pre-qualified bidders can place a bid."). The ways in which agents may interact is enhanced through a rich set of action-related constructs that our framework provides, which account, amongst others, for conditional and non-deterministic effects. We also define *transactions* as a means to safely bundle actions together, allowing agents to simultaneously trade goods, issue rights, and take on obligations.

Unique to our framework is that we take a *black-box* approach to the specification of agents and impose no restrictions on their design and internal workings. As a result, the monitoring system does not need to perform complex activities such as *planning* on behalf of agents, or solving hard winner-determination problems in auctions. The monitoring system is instead required to *verify* whether certain goals are established, by having agents state obligations and then provide sufficient information to the monitoring system to enable their easy verification. The task of verification

is assigned in our framework to a *master agent*. An agent in the role of an auctioneer can, for instance, provide market-clearing prices to allow the master agent to verify that the former has met its obligation in regard to solving the winner determination problem optimally, but without requiring the master agent to solve the optimization problem itself. Appropriate punitive sanctions may be taken by the master agent if the specified condition is not met, as defined in the obligation. This is a middle road between a completely formal but hard to program system, and a completely open-ended but informal system. Obligations, and also rights, provide a well-defined interface between the monitoring system and the agents that act in the market.

1.1 Design Principles

Our framework is fully implemented, and designed to respect four central principles. The first two principles exemplify the generality of the proposed framework, while the other two ensure soundness in monitoring and enforcing rights and obligations, and in determining the effects of executing an action.

Black-box Principle: Agents are entities that exist outside our framework and can be implemented in a different language (or be people, interacting through proxy agents). Agents reason based on their own beliefs and this reasoning is completely decoupled from the framework that monitors the evolution of the economic environment world in which the agents participate.

Free-will Principle: Agents choose which action to take and cannot be forced to take actions. Rather than require an agent to take a specific action, for instance an action that satisfies its obligations, the framework monitors an agent's obligations and is empowered (via the master agent) to execute the sanctions associated with an obligation when an agent fails to meet it.

Restriction Principle: The monitoring system is able to track the rights of agents and restrict the execution of actions for which an agent does not hold appropriate rights.

Soundness Principle: When an action is invoked, and if the appropriate right is held and the action's executability preconditions are met, then the action's effects are produced in accordance with the laws of the economic environment.

By the black-box principle, an agent retains its autonomy in terms of the state of its beliefs and the reasoning mechanism it employs — the framework does not impose any requirements on the internal workings of an agent, nor does it force an agent to reason in a prescribed way, other than its ability to interact with the provided interface. By the free-will principle, the agent freely chooses which actions, among those available, it will take. We note that although related, the black-box and free-will principles are distinct. It would be possible to design a framework where either of these two principles would hold, but not the other. The former principle ascertains only that the internal workings of an agent are unknown, leaving open, however, the possibility that the agent might be expected to act in a prescribed manner without any free-will. Similarly, the latter principle ascertains only that the agent is able to freely choose its next action, leaving open, however, the possibility that the agent's internal state is visible to the system that monitors the economic environment (and perhaps to the rest of the agents).

In contrast to the privacy and freedom of choice offered by the first two principles, an agent's obligations are monitored, and any (presumably punitive) actions are executed upon the world

entering a state that indicates a failed obligation. By the restriction principle, an agent may only *request* that an action is taken. These requests are screened by the monitoring system and actions are executed only if an agent has the appropriate rights.

This approach of restricting action executions complements the free-will principle: an agent's available actions can be restricted by the monitoring system, but the agent still retains the choice of which (if any) actions to take. The soundness principle requires that the monitoring system will always respect the laws of the market, as defined by the designer of a domain. The practical implication of this principle is that the monitoring system acts as a trusted party; its actions can be independently verified, since the monitoring system's specification and implementation is open for inspection by all involved parties.

We illustrate our design principles by analogy to the `ebay.com` marketplace. eBay participants freely choose when to enter or leave eBay's economic environment. Once a participant has joined the market (and has a valid user ID), then she can interact with the market via the WWW interface. For instance, as a seller this occurs by initializing a new auction and providing information about the good for sale. This will typically obligate the seller to enter into a contract to sell the good to the agent associated with the highest bid received by the auction deadline. A buyer interacts with the eBay market through an eBay proxy agent, to which a buyer reports a (maximum) willingness-to-pay for the good. A valid "bid" action of this form is one that is higher than the bid price posted for the current winner. Thus, the right to take such an action depends on the current state of the auction. Following a valid bid, the proxy will compete with other bidders within the auction either until the participant is winning or until her maximal willingness-to-pay is reached. The participants then observe the new state, and while the auction remains open participants can continue to revise their bids upwards.

The black-box principle applies to eBay. Participants on eBay ("agents" elsewhere in this paper) are independent of the eBay marketplace and autonomous. The only requirement placed on a participant is that she can interact with eBay's market through the interface, e.g., via web page links and forms. The eBay market is also consistent with the free-will principle. eBay does not (and cannot) force a participant to honor a transaction. Rather, eBay encourages others to punish a participant that fails to meet an obligation by providing the other party in the transaction with the right to leave negative feedback in the recommendation system.[1] We can also see the restriction principle: eBay participants can auction items, or bid on items, but only when they have appropriate rights. As noted above, a participant only has the right to submit a new bid to the proxy if the new bid is high enough. Lastly, the soundness principle applies, for instance, in that a paying agent is guaranteed that if the paying action is invoked via an electronic cash system such as `paypal.com`, and the action's execution preconditions are met, then the appropriate effects will be produced, irrespective of what other events (e.g., the concurrent execution of some other payment, or the closing of some auction one hour ago) take place.[2]

[1] From `http://pages.ebay.com/help/confidence/programs-investigations.html`: *"eBay cannot force a seller to honor their transactions. You should leave appropriate feedback for the reluctant seller [...]".*

[2] If there are insufficient funds in the payer's bank account, then the action execution will not transfer funds to the payee. The fact that the goal of executing the action was not met is orthogonal to the soundness principle, which states that the effects of the action, whatever those may be depending on the state of the world, will be produced.

1.2 Applications

We envision three main applications of our framework for specifying and monitoring economic environments using rights and obligations:

- The scripting language can be used by itself, either as a means for describing economic environments (and their rules) in an agreed upon formal syntax, or as a specification language for contracts amongst individuals, agents, or businesses. The language's semantics guarantees that the descriptions are unambiguous, while the natural syntax and semantics allow human participants to understand and reason about the described markets. Arbitration in the case of disagreement on the provisions of a contract can be done through the monitoring system, which can provide objective facts about some disputed issues.

- The framework can be used to provide a platform for prototyping, testing and evaluating newly developed automated agents and market protocols. A market protocol can be described in the scripting language, and the monitoring system can then be used to run a simulation of the market. Agents can participate in the economic environment, interacting according to their specifications. The entire history of the market is recorded, allowing for subsequent analysis of the performance of the market and agent designs. This evaluation process can support the testing of automated agents and the testing of market protocols before they are deployed, and hence allows a designer to anticipate or prevent possible shortcomings of the developed designs.

- The framework provides the basis for deploying an electronic market. The monitoring system can be used as the underlying engine that keeps track of the market evolution, and can ensure that the market laws, as described in the scripting language, are adhered to by all the participants; e.g., with rights verified and with punishments associated with unmet obligations executed. This engine can reside on a server with appropriate web-based interfaces to allow human users and automated agents to interact with the market; e.g., with both web browser and API interfaces. The history recording feature can be used to verify that the correct transactions took place in the electronic market.

Using the framework for the deployment of electronic markets seems to carry with it some inherent difficulties because participants do not act only within the purview of the electronic market, but can freely interact — and also be required to act — in other ways. One question that naturally arises is how are goods and money "transferred" between participants in the physical world? For example, suppose that a car is sold in the electronic market, for the amount of US $5000. How can one ensure that the car will also be sold in the physical world and that the buyer will pay the seller?

The described problem is not specific to our framework, but rather a general problem faced by any electronic market. One response is to couple the electronic market directly with transaction execution in the physical world. For instance, the `amazon.com` electronic marketplace enables this by providing third-party logistics, e.g., with sellers able to store goods in Amazon's warehouses and entering into a contract with Amazon to ship goods upon the completion of a sale in the electronic marketplace. Another approach is to build the rules of the electronic market on top of laws in the physical world. The `ebay.com` marketplace provides an example of this approach, with all actions in the electronic market treated as binding contracts under national legal jurisdiction. Thus, by agreeing to sell her car on eBay, a seller also enters into a real world contract, enforceable by the

laws of the country, to sell her car in real life. If this is the approach followed, then the semantics adopted for the virtual economy should be rich enough to serve this dual role so that the virtual world "bootstraps" onto the real world. The formal semantics of our framework, and the history of all actions and states, play an essential role here, in providing an unambiguous language for contracts and a record of actions invoked by participants.

1.3 Related Work

The important role that property rights play in well-defined economic environments is well understood and much discussed in the foundational economic literature on market institutions and organization theory. For instance, Tirole [55] writes,

> "*A decision right or authority granted to a party is the right for the party to pick a decision in an allowed set of decisions. A property right on an asset, i.e., its ownership, is a bundle of decision rights.*"

As discussed by Hart [26], it is standard to model a firm as a collection of assets, and consider the ability of a firm to retain a specific subset of its bundle of rights while selling all other residual rights [26]. We provide this kind of expressiveness in our formal semantics. Moreover, the role of obligations and commitment is recognized to be important for writing efficient contracts [26], and also in the design of economic mechanisms such as auctions, where obligations provide constraints that enable an agent to commit to the use of a particular rule in determining the outcome of a negotiation process [28].

The theory of *deontic logic* and *normative systems*, as described by McCarty [37] and in the edited collection of Meyer and Wieringa [38], provides the logic of rights and obligations, and is concerned with performing inference about what *should* happen in a system while still allowing for the possibility of non-normative behavior, for example seeking to establish the validity of statements such as "Is every obligatory action permitted?"; see also Carmo and Jones [10]. For a survey of applications of deontic logic within computer science, see Wieringa and Meyer [62]. We adopt *soft* obligations, with sanctions imposed on agents in case of failure to meet their obligations. The alternative approach of adopting hard obligations is inconsistent with our free-will principle that we adopt here; agents in our environments may well take actions that lead them to states in which their obligations are violated, for instance when striking a tradeoff between local goals and sanctions.

By adopting soft obligations we also avoid certain well-known paradoxes discussed in the deontic logic literature. Such paradoxes occur, for instance, in the presence of "contrary-to-duty" obligations [38, 68], that is, secondary obligations whose provisions hold in case a primary obligation is violated. If it is known that the primary obligation is violated, then one is lead to deduce that the provisions of both primary and secondary obligations apply, even if these provisions are incompatible; this results in an inconsistent state of affairs when the employed obligations are hard (in our sense). On the other hand, if the employed obligations are soft, then no inconsistency is reached *per se*. Instead, one is lead to deduce that an agent is expected (but not forced) to reach a state where both obligations are satisfied. As initially assumed, the primary obligation is necessarily violated, but this does not preclude (through an inconsistency) the agent from pursuing the satisfaction of its "contrary-to-duty" obligations; the intuitive interpretation of such domains is thus preserved.

Prior work in multi-agent systems has considered the role of rights and obligations for the specification and semantics of *open systems* through *electronic institutions* [4, 15, 16, 44, 53, 64]. Such systems allow agents to enter and perform tasks, while providing an explicit specification of

norms that enable reasoning about the consequences of failure [17, 58]. López y Lopez et al. [35] note,

> "*[. . .] the introduction of norms that help to cope with the heterogeneity, the autonomy and the diversity of interests among autonomous agents has been considered as a key issue towards the computational representation of open societies of agents.*"

This prior work differs in terms of whether obligations place hard or soft constraints on agents. Along with hard constraints comes the need for the total control, or *regimentation*, of agents [8, 22, 25], together with methods to verify compliant agent protocols [2] and validate planned actions [67]. Soft constraints, on the other hand, allow agents to autonomously decide whether to comply and how to act [4, 16]. Recent work generally adopts our philosophy that autonomy will typically preclude hard constraints, due to private agent states and goals and also agent autonomy in taking actions [1, 35]; indeed, Fornara and Colombetti [20] note that regimentation is often impossible and sometimes detrimental.

Approaches differ in whether the monitoring system actively enforces sanctions, perhaps through controllable agents, as in our work and many others [1, 4, 6, 9, 16, 19, 21, 35, 57], or only passively maintains the global state and informs agents of their obligations and the failed obligations of other agents (with an appeal to "social control") [15, 56]; see Castelfranchi et al. [11] for an earlier discussion. López y Lopez et al. [35] also adopt the idea of "promoters" that can provide positive rewards as a complement to the negative consequences of sanctions. Obligations in our framework can trivially be used to implement promoters.[3]

A feature that we share with most of the literature is that we provide, through rights, or the absence thereof, for *prohibition* and *permission* on (complex) actions; see for instance Wyner [66]. On the other hand, the conditions in our obligations are state-based (cf. [15]) rather than action-based (cf. [4, 8]); see d'Altan et al. [14] and Wyner [66] for a further discussion of *deontic action logics* in which obligations are associated with actions and comparisons between this "ought-to-do" approach and our "ought-to-be" approach. Our language does in fact permit simple action-based obligations, when the invocation and the successful (or not) execution of actions can be encoded in states. For instance, an auctioneer may be obliged to take a "sell" action and by so doing move the world into a state in which an item is sold to the highest bidder. We generally agree, however, with López y Lopez et al. [35], who observe that normative goals — equivalent to associating obligations with states — are *more compatible with autonomous agents who can choose to satisfy goals instead of being told how to do it. . . .* An extreme, opposing approach seems to be provided by *Deontic interpreted systems* [34], in which agent logic is directly validated by, and visible to, the electronic institution.

To the best of our knowledge, this is the first work to adopt rights and obligations as *first-class goods* that agents can explicitly trade and exchange. The ability to sell bundles of rights, and limit them with obligations, seems crucial to the functioning of markets as defined by authors such as Tirole [55] and Hart [26]. In our framework, transactions enable the safe bundling of rights and obligations to make such exchanges possible. Our approach is significantly more general in this regard than earlier work [15, 53], in which it is simply observed that agents might contract with

[3]Our obligations have three arguments. When interpreted as (satisfy, violate, action) then this is an obligation, and the action is expected to be punitive. In order to encode promoters, one need only treat "violate" as the goal, "satisfy" as the expiration of the offer, and "action" as the reward. That is, promoters are simply obligations that an agent is actively trying to "violate", so as to cause the invocation of the action, which is expected to be beneficial.

other agents to satisfy the formers' obligations. In terms of auction semantics, while Wurman et al. [65] provide a formal taxonomy for the rules of electronic auctions, they provide a semantics for high-level auction attributes rather than building up market protocols from underlying principles related to rights and obligations.

Similarly, we are unaware of any prior work that explicitly sets out to model the *rights that derive from goods in economic worlds*, or the semantics of *ownership and possession*. Indeed, while many authors consider the design of open agent societies, and formal semantics for electronic institutions and organizations [3, 13, 18, 19, 24, 42, 59], the emphasis seems different from our work. To illustrate some differences, we can consider the work of Arcos et al. [3], which introduces an electronic institutions development environment (EIDE). (Notably, the methods of Arcos et al. [3] have been applied to the deployment of an electronic market for fish trading [13].) We share some features with EIDE [3], such as ignoring the internal details of how agents make decisions, and the use of a monitoring system and special "mediating" agents (their Institution Manager, our master agent). On the other hand, Arcos et al. (and similarly [24, 42, 59]) seek to model institutions such as market protocols at a much greater level of detail than in our work. It is typical to adopt process algebraic approaches and concurrency theory to model the detailed workflow of protocols. In return for this, the *verification* of some aspects of correctness (such as the reachability of states, liveness, etc.) is possible. By adopting rights and obligations as the language by which commitments are made between designers, agents and our monitoring system, we allow for a more lightweight, flexible, and open approach.

Compared with the work of Vazquez et al. [59] on an Organizational Model for Normative Institutions (OMNI), our approach is not concerned with *societal* structure and does not consider agent roles or hierarchies. Neither are we concerned with the meta-problem of how agents can negotiate new social norms, or the stability of social norms [7, 54]. Finally, our notions of conditional and limited rights are shared with previous work on formal specification languages for financial contracts, namely that of Peyton Jones and Eber [29], although that work focuses on the formal description and analysis of new forms of financial contracts and not on providing frameworks for the description, simulation, and construction of open agent societies. Similarly, while the π-calculus has been used for the specification of a complex model of a Spanish fish market by Padget and Bradford [43] (see also Rodriguez-Aguilar et al. [45]), the goal in that work was to assist with the development, design and analysis of complex institutions, rather than monitor and enforce properties of dynamic state.

1.4 Paper Outline

Section 2 provides an overview of the scripting language and monitoring system, and introduces the semantics of our model. In Section 3 we introduce and discuss the role of rights and obligations. Section 4 defines and justifies the fundamental *axioms* provided in our framework and relates them to the standard notions of ownership and possession. Section 5 provides some implementation details. A number of detailed examples are used in Section 6 to illustrate the way in which rights and obligations can be used in application to various market mechanisms. We conclude in Section 7.

2 Architecture and Model Semantics

Our framework consists of a *scripting language* and a *monitoring system*. The scripting language provides the necessary syntax for describing economic environments and the monitoring system provides the language semantics. This is analogous to the case of programming languages that are accompanied by operational semantics; a programmer uses the language to write a program, while the semantics of the program is defined through the program's execution in a prescribed manner. Our programmer is the *domain designer*, and the program is the *domain description*, a collection of laws governing the particular economic environment being modeled. The agents themselves are also programmed by some programmer, but this is performed outside of our framework.

Before we delve into the details of our framework, we find it useful to discuss some issues pertaining to the guarantees our framework can provide with respect to its semantics. As in most programming languages, it is not possible to guarantee that any domain description will yield any reasonable behavior. In particular, we cannot a priori guarantee that an economic environment being modeled will respect any liveness or safety properties, that deadlocks will be avoided and progress will be made, or that some unwanted or unintuitive behavior will not occur. All these events are possible, and are determined only by the domain description that is fed into the monitoring system. In fact, it is impossible to even provide conditions under which a domain description would avoid such unreasonable behavior, since that would constitute a solution to the Halting Problem, which could be encoded in a domain description (since the proposed scripting language essentially extents Prolog, a Turing-complete programming language). The burden of ensuring that an economic environment proceeds as expected lies entirely on the domain designer.

2.1 The Scripting Language

The scripting language is built on top of Prolog, and enjoys its powerful semantics and rich syntax. Thus, natural constructs appropriate for describing markets, such as lists of objects, predicates defining attributes of objects, and general schemas that unify with specific instances, are all present in the scripting language. Furthermore, the language is easily extensible, allowing the introduction of new constructs through the addition of Prolog code within domain descriptions. The monitoring system is also implemented in Prolog, which provides a clean way to interpret domain descriptions and run the corresponding economic environment.

The domain designer can import libraries describing economic market laws that are commonplace in a variety of settings. This is analogous to ordinary programs, which can typically import libraries that provide specific functionality. We have written a number of such libraries, including: a library on "exchanges of goods" with laws on how goods can be traded, given, or sold between agents; a library on "handling rights and obligations" with laws on how rights can be given up, issued, or revoked, and laws on how obligations can be taken on, imposed, or cleared.

2.2 The Monitoring System

The architecture of the monitoring system, and its interface with the agents, is shown in Figure 1. The monitoring system runs a virtual economic environment, as governed by the laws specified in the domain description provided by the domain designer. The laws define the initial state (e.g., an allocation of goods), the objects that populate it, and the relevant attributes of these objects. The laws also dictate how agents might join or leave the market (e.g., by specifying that each agent is

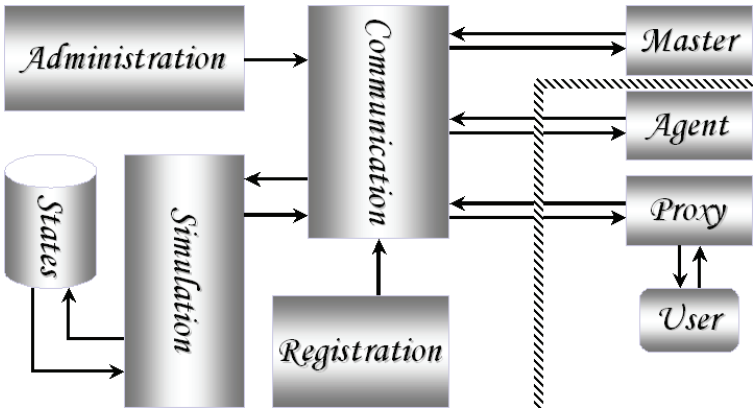

Figure 1: The modules of the monitoring system.

granted a certain amount of money when entering the market), and the available actions through which the agents might affect and observe the market's status.

The agents are not simulated as part of the virtual world and all local deliberation remains private to an agent. Rather, an agent makes decisions independently and acts through communication with the monitoring system. The monitoring system executes actions only if an agent has the appropriate rights. Success or failure of actions is recorded, and the state of the environment is updated accordingly. Periodically, the monitoring system checks whether an obligation has been satisfied or violated, recording the event and exercising the appropriate punitive sanction in the case of a state that indicates a violation.

Note that even if a particular set of agents is fixed, a given economic environment may still yield multiple possible sequences of states, each sequence describing an evolution of the virtual world. The sequence among all possible ones that will actually occur depends on the outcome of stochastic events within the environment and also local to each agent. Each such sequence is called a *scenario* and corresponds to a specific instantiation of an economic market. Although the possible scenarios are dictated by the domain description, the actual scenario that occurs is ultimately defined by the interaction between the environment and the agents.

The monitoring system is initialized by the *administrator* of the system, who is responsible for setting up its different modules. Each module is an independent process, which can reside on its own machine and communicate with the rest of the processes in a server-client manner. The administrator[4] selects the domain description to be loaded and sets up the market's initial state through the master agent, a special trusted agent defined within our framework. From that point onwards the monitoring system awaits for agents to join the market and request the execution of actions.

The *communication module* provides the channel through which agents can invoke actions. The *simulation module* is responsible for handling the execution of the invoked actions, updating the state of the market, and reporting new state information to agents. The administrator can also intervene and instruct the master agent to execute specific actions. Our framework also allows the use of *proxy agents* that can be used by human users who wish to interact with the market. The

[4]The system administrator is responsible for the execution of a particular market, and need not be the same person as the domain designer for that particular market. Indeed, we think of domain designers as people who may write domain descriptions, perhaps for a fee, independently of how these domain descriptions end up being used.

implementation of the system includes an implementation of a simple proxy agent ready for use.

In addition to allowing for intervention by the administrator, the master agent is used to capture exogenous events that are outside the agents' control. For instance, the initial state of the system is populated by means of the master agent executing the `initialize` action once the domain description is loaded. The domain designer can specify the effects of the master agent's interventions, for instance in situations such as the arrival or departure of an agent, the passing of time, and the execution of actions by agents. The master agent is restricted by the designer to execute only a certain fixed set of actions, and on a well-defined and pre-specified set of occasions. The existence of a master agent within our framework is consistent with the role of central, impartial, and trusted authorities in human markets, for example as provided through the laws and courts of a country. Similar to the role of a court in punishing violators, the master agent is responsible for executing the punitive sanctions associated with obligations if violations occur.

2.3 Model Semantics

In this section, we describe the semantics of the dynamic model of our monitoring system. The world goes through a sequence of states, with each state specifying values for the attributes of the objects that populate the state. We discuss what these objects look like, how their attributes can be modified, and how they define the state of the market.

2.3.1 Objects and Classes

Each domain defines a set of classes (as in object oriented programming), each associated with a set of attributes. Objects are instances of classes.

Definition 1 (Classes) *Let*

`classes([#class, ...])`

denote the set of classes in a domain description. Each class `#class` *is of the form*

`(#name, [#attribute, ...])`,

where `#name` *is the class name, and* `#attribute` *is the name of one of the class attributes.*

For example, the following code snippet defines that the classes of the domain description contain the class for apples, and specifies that instances of this class (i.e., actual apples) have attributes defining their owner, their possessor, and their weight:

`classes([..., (apple, [owned_by, held_by, weight]), ...]).`

A set of basic classes is defined in our framework. The domain designer can also extend this set depending on the market being modeled.

An object defines a list of attribute-value pairs. One of these pairs corresponds to the attribute `instance_of`, and the associated value is the name of the class of which the object is an instance. The rest of the pairs correspond to the attributes of the object's class and the values associated with these attributes. In addition to this list, every object also defines a unique name, which is used by the agents and the monitoring system to reference that object.

Definition 2 (Objects) *Let*

`(#name, [(instance_of, #class), (#attribute, #value), ...])`

denote an object, where `#name` *is the object name,* `#class` *is the class of which the object is an instance,* `#attribute` *is an attribute of the class* `#class`*, and* `#value` *is the value that object* `#name` *associates with attribute* `#attribute`*. All attributes of the class* `#class` *should appear in the definition of the object* `#name`*.*

Thus, an apple object is a structure of the following form:

`(#name, [(instance_of, apple), (owned_by, #owner), (held_by, #possessor), (weight, #weight)])`.

Note that the notions of *ownership* and *possession* are readily supported as attributes of objects. Transferring ownership or possession of an item from some agent to another reduces to simply changing the values of the item's attributes `owned_by` and `held_by`.

In a similar fashion, we define a class for *accounts*. One of the attributes of this class is the `balance` attribute. Each account object defines a value for this attribute, which corresponds to the amount of money the owner of the account has in the account. Transferring money through payments is implemented by changing this balance in an appropriate way. A similar treatment can be employed by a domain designer for defining "containers" for objects, when only the quantity and not the explicit representation of certain objects is important, providing, thus, a level of abstraction. When trading stock shares, for instance, an agent may be given a portfolio object that keeps track of the number of the held shares, without needing to identify each share as an individual object.

A special class defined in our framework is the `event` class.

`classes([..., (event, [#description, #happened_at, #expired_at]), ...])`.

As the name suggests, objects that are instances of this class serve to record the various events that take place in the virtual market. Such events include the initialization of the market, the arrival or departure of agents, the actions invoked by agents (including sanctions invoked by the master agent), and the instantiation of new states as a result of preceding events, or of the passing of time.

The use of objects and classes provides a uniform treatment for both *physical goods*, like apples, and abstract goods, like rights and obligations.

2.3.2 States

The *state* of a market represents all information that exists in the environment of the agents. This includes the set of objects along with values for their attributes, as well as the set of agents that populate the market:

Definition 3 (States) *Let*

`state(#agents, #objects)`

denote a state, where `#agents` *is a list of agent names, and* `#objects` *is a list of objects.*

States are constructs that are manipulated by the monitoring system, and are not explicitly represented in a domain description. Note that the state of a market does not include information about the internal states of any of the participating agents; following the black-box principle this

information is not available to anyone but the agents themselves, which operate outside the monitoring system and only communicate with it. Nonetheless, the presence of an agent in the market, and the ownership or possession of goods, rights, or obligations by an agent is information that is available to the monitoring system. Whether this information is also viewable by other agents is an orthogonal issue that we discuss later.

2.3.3 Actions

The existence of objects in the state of a market, and the values of the attributes of objects, are only affected by means of actions invoked by the agents through their interaction with the monitoring system. The master agent, controlled by the monitoring system, is also able to invoke actions.

In their *primitive* form, actions have preconditions and effects:

Definition 4 (Primitive Actions) *Let*

```
action(#agent, #action) :- preconditions(#preconditions), effects(#effects)
```

denote a primitive action `#action`*, with a list of preconditions* `#preconditions`*, and a list of effects* `#effects`*.*

When the monitoring system attempts to execute an action `#action`, following its *invocation* by an agent `#agent`, it first checks whether the agent holds an appropriate right, and whether the preconditions `#preconditions` of the action are satisfied, and subsequently updates the state according to the action's effects `#effects`.

An action's preconditions, which can depend on the agent `#agent` that invokes the action, are those conditions that need to be met for the action to be "physically" executable. For example, an action transferring funds from some account is conditioned on the account containing the corresponding amount. An action opening an auction on some item is conditioned on the existence of that item. Action preconditions may contain any Prolog predicate, whose satisfiability will be verified before the action is executed. Among the available predicates is the special `value(#object, [(#attribute,#value),...])` predicate offered by our framework, which is satisfied exactly when object `#object` assigns the value `#value` to the attribute `#attribute`, for every `(#attribute,#value)` pair in the list.

The possession of an appropriate right to execute an action is treated as an *implicit precondition* of every action. The monitoring system takes care of this without the need for the domain designer to explicitly add such a precondition. *We emphasize that a critical difference between the explicit preconditions of an action and the implicit precondition of holding an appropriate right, is that the latter can be traded as a result of the treatment of rights as goods.*

An interesting question arises, following the fact that rights can themselves be conditional. With a conditional right, then both the right's preconditions and the action's preconditions must be met for the action to be successfully executed. How should the domain designer choose whether any particular condition is to appear in the action's preconditions or the right's preconditions? A rule of thumb that we suggest is to ask whether the precondition is a "physical ability" or a "legal allowance" precondition; the former are preconditions of the action, while the latter are conditions of the right.

Intuitively, when conditions are placed on an action it is because whenever such conditions are not satisfied, the action can not be executed under *any* circumstances. The action of giving an item

has the precondition of holding that item; if this condition is not true, it is "physically" impossible for the action to be executed. On the other hand, conditions on a right leave open the possibility that someone with a less constrained right could in fact execute the action. For instance, the right to execute the action of selling an item is usually conditioned on owning the item. Yet, one can easily imagine situations where an attorney, for instance, is granted (by the item's owner) the less constrained right to sell the item without the attorney owning it. Thus, the condition of owning an item is a "legal allowance" and should not appear in the action's preconditions.

This distinction between physical preconditions and rights is somewhat less clear when one models electronic markets, since certain actions lose their physical aspect. As an example, imagine the precondition of placing a bid in an English auction that states that the bid should be higher than the current highest bid. Is this a precondition of the action of placing a bid, or a condition on the right to execute the action? In most cases the answer to such questions is inconsequential. It is up to the domain designer to provide a proper answer when the choice is important.

2.3.4 Effects

Each primitive action is defined to have a set of effects that it produces when the action is successfully executed, and which can depend on the agent `#agent` that invokes the action. The effects are produced sequentially, and in particular, preceding effects can change the state in which subsequent effects are executed. Our framework offers a set of effects that provide enough expressiveness in a variety of situations:

Definition 5 (Effects) *Let*

`create(#object,#class)`, `destroy(#object)`, *and* `set(#object,#attribute,#value)`

denote respectively the action effects that create object `#object` *as an instance of class* `#class`*, destroy object* `#object`*, and set the attribute* `#attribute` *of object* `#object` *to the value* `#value`*.*

These three basic effects (`create`, `destroy` and `set`) are augmented by appending a condition to an effect, with

```
#effect where #condition,
```

where condition `#condition` can be any Prolog predicate. The intended semantics of a conditional effect is then that the condition is checked under Prolog semantics, and for every distinct way the condition is satisfied (i.e., every instantiation of the Prolog variables that `#condition` contains), the effect `#effect` is produced. Since the effect may share Prolog variables with the condition, this simple construct allows one to obtain many different and interesting effects.

In the simplest case the structure allows for simple conditional effects, by having conditions that are satisfied at most once and do not share free Prolog variables with the effect. The following code snippet corresponds to defining the effect of the `turn_key(car)` action that results in the car engine running if the fuel tank is full:

```
set(car, engine, running) where value(car, [(fuel, full)]).
```

This example also illustrates the most typical use of conditional effects, where conditions test whether certain objects have certain values for their attributes. Things become more interesting when the condition and the effect share some free Prolog variables. In that case, we get a different

effect according to how the variables in the condition are grounded. Hence, we can possibly get non-deterministic behavior from an action, since the effects it produces might differ according to the state. For example, one can represent the effect that one's account balance increases by the amount written on the deposited check as follows:

```
set(account, balance, Balance+Amount) where
(value(account, [balance, Balance]), value(check, [(amount, Amount)])).
```

Recall that the condition is any Prolog predicate. In particular, the Prolog predicate `(#predicate, ..., #predicate)` is satisfied exactly when all listed predicates are satisfied.

Allowing the conditions to be satisfied more than once leads to quantifiable effects; one can, for example, define an effect that gives another agent all the possessions of the agent executing the action, by having the effect be repeatedly produced, once for each grounding of the condition:

```
set(Object, held_by, alice) where value(Object, [held_by, bob]).
```

Finally, one can even introduce probabilistic effects, by having the condition contain a random process. To this effect, our framework offers the `choose(#choice, #distribution)` predicate, that when checked as part of some condition compares/assigns `#choice` to a probabilistically chosen value. The possible values and their associated probabilities are defined by the list of pairs `#distribution`. Thus, for example, the following code snippet corresponds to defining the effect of the `flip(coin)` action that probabilistically produces either heads or tails:

```
set(coin, side_up, Side) where choose(Side, [(0.5, heads), (0.5, tails)]).
```

As per the specification of the `choose(#choice, #distribution)` predicate, Prolog will ground `Side` to `heads` with probability 0.5 and to `tails` with probability 0.5.

Fixing `#choice` to a specific value results in the predicate being satisfied probabilistically. The following code snippet corresponds to defining the effect of the `drop(plate)` action that with high probability results in the plate being broken (otherwise it remains in its current status):

```
set(plate, status, broken) where choose(true, [(0.95, true), (0.05, false)]).
```

Prolog attempts to unify `true` with `true` with probability 0.95, in which case the unification succeeds, and the effect of the plate becoming broken is produced. With probability 0.05, Prolog attempts to unify `true` with `false`, which fails, and thus no effect is produced; the status of the plate remains unchanged.

2.3.5 Transactions

In their *transactional* rather than primitive form, actions are ordered sequences of actions (which may be primitive, or transactions themselves).

Definition 6 (Transactions) *Let*

```
action(#agent, #transaction) :- transaction(#actions)
```

denote the transaction `#transaction` *comprised of the list of actions* `#actions`.

When executing a transaction either all or none of the constituent actions are successful. The execution model employed by the monitoring system is to execute each of the actions in turn, updating the world state after each action. If any of the actions fails to meet its preconditions, or the agent fails to have an appropriate right at the time of each constituent action's execution, the *entire* execution is rolled back to its original state.

We take the approach that a transaction invoked by an agent will be executed if the agent either has the right to execute the transaction as a whole, or it has the right to execute each of the transaction's constituent actions independently. In the second case, the right for each particular action should be held at the time that the action is considered, once all of its preceding actions have been executed. It is possible, then, that the first action may grant or revoke an agent's right to execute a subsequent action in the transaction, making transactions a very expressive modeling device.

The notion of transactions is a powerful one, with a number of applications, such as that of implementing safe exchanges of goods. The transaction:

```
action(bob, sell(apple,1,alice)) :-
transaction([give(apple,alice), take_money(1,alice)]),
```

for instance, specifies a fail-safe way for Bob to sell his apple to Alice for US $1, without either party being vulnerable to the other's reneging. Assuming Bob has the right to execute the transaction, the monitoring system will allow Bob to execute the actions `give(apple,alice)` and `take_money(1,alice)` which constitute the transaction. Nonetheless, the right to execute the transaction does not imply that Bob has the right to execute the constituent actions independently of each other, and out of the context of the `sell(apple,1,alice)` transaction.

Transactions can also be used in other ways. For example, a domain designer might use transactions simply as a way of providing synonyms or slight variations of actions, by having only a single action in the transaction, and possibly fixing some of its parameters. Thus, the following transaction can be used to define the `donate` action that sells an item for the token price of 1 cent:

```
action(Agent, donate(Item,Receiver)) :- transaction([sell(Item,0.01,Receiver)]).
```

To make things even more interesting, actions within a transaction are allowed to be conditioned in a way similar to that of conditional effects. It is then straightforward to define the `pay_and_tip` action that can be used to pay a restaurant bill with a 20 percent tip:

```
action(Agent, pay_and_tip(Amount,Receiver)) :-
transaction([pay(AmountWithTip,Receiver) where AmountWithTip = Amount * 1.20]).
```

Using the same construct, a transaction can be very easily defined that states that all actions with a certain property (or within a certain set) should be executed. The following transaction defines the action `sell_all` that sells all items within a list, each for US $2. The condition of the `sell` action used within the transaction uses the Prolog predicate `member` to traverse all elements of the list `Items`; recall that as in the case of conditional effects, the conditional action is executed once for each grounding of the condition.

```
action(Agent, sell_all(Items,Receiver)) :-
transaction([sell(Item,2,Receiver) where member(Item,Items)]).
```

Another interesting use of conditional actions is that it allows the domain designer to define transactions that correspond to contingency plans, where actions are executed depending on the conditions that hold in the given state of the market. The following action, for instance, lets an agent sell an apple and an orange that the agent may or may not hold.

```
action(Agent, sell_fruits(Receiver)) :- transaction([
  sell(apple,1,Receiver) where value(apple, [held_by, Agent]),
sell(orange,1,Receiver) where value(orange, [held_by, Agent])]).
```

We emphasize that even if only the orange is held by the agent, the transaction will not fail, since the constituent action of selling the apple is conditional on the agent holding it; if the agent does not hold the apple the conditional action is trivially satisfied, and the second constituent action of the transaction will be executed.

Transactions can also be used to bundle the issuing of rights and the taking on of obligations. Examples of this will be seen in the context of various auction mechanisms in Section 6.

2.3.6 Built-In Actions

The framework provides a set of built-in actions and transactions. Two such actions are those for creating objects and setting their attributes to specific values. These actions are available to the master agent to instantiate objects in the domain. Another action available to the master agent is that by which an agent loses all of its rights, or is banned from the market. Additional built-in actions, available when an agent has the appropriate right, include:

- giving ownership and/or possession of goods to other agents, or giving up ownership and/or possession of goods;
- taking ownership and/or possession of goods from other agents, or taking on obligations;
- issuing ownership and/or possession of rights to other agents;
- giving money to, or taking money from, other agents;
- exercising an expiring right, by executing the intended action and then giving up the right.

Of course, these built-in actions are not meant to be the only actions that are available to agents. In addition to the built-in actions, the domain designer can define actions specific to the domain being modeled, like, for instance, the `fill_tank(#car)` action whose effect is that of setting the `fuel` attribute of the `#car` object to `full`.

2.3.7 The Query Action

Important in every economic market is the notion of *private information*. Consider, for example: a sealed-bid second-price auction, where the attribute representing the collected bids must be viewable only by the auctioneer; or, situations where an agent might, or might not, have the right to see whether a second agent has the right to perform an action, or even whether the second agent has the right to see certain attributes of the market.

Privacy of local state is provided in our framework by treating the values of all object attributes as hidden by default. An agent can seek to obtain the value of an attribute by invoking the `query`

action, but an agent must first have the right to query an attribute if the action is to be successfully executed:

Definition 7 (Query) *Let*

```
query(#object,#attribute)
```

denote the action of querying the value of attribute `#attribute` *of object* `#object`*. When the action is executed, the agent that invoked the action learns the queried value.*

Unlike other actions in our framework, the `query` action does not directly affect the market being simulated; instead it informs the participating agents about the state of the market, and the agents may then take other actions that will affect the market.[5]

Due to the centrality of the `query` action, the monitoring system automatically executes the action on behalf of all agents whenever the agents have the right to do so, and informs them of the values of the queried attributes. An obvious exception to this rule is when an agent has the right to query an attribute only as part of a transaction; the monitoring system will not exercise the right to execute such a transaction, allowing the agent to choose when the transaction is to be executed.

The `query` action also provides a method for communication between agents. Our framework supports the creation of objects that are instances of the special `info` class. These are objects that all agents have the right to create, and whose sole purpose is to store pieces of arbitrary information. Creating these `info` objects is another example of a built-in action. Each `info` object has a `value` attribute, that the owner of the object can assign to any arbitrary value. By giving another agent the right to query this attribute, the former agent can selectively (and secretly from the rest of the agents) communicate information to the other agent.

3 Rights and Obligations

Central to our framework are rights and obligations. We adopt the approach that rights and obligations are treated as any *first-class* object with a set of predetermined attributes whose values are part of the market's state. Rights and obligations are tradable goods that can be given, taken, exchanged, sold, and so on.

To give a taste of why this idea is very powerful, consider an audio compact disk sold in an eBay auction, with the winner being awarded the item as per eBay's rules. What is implicit in this transaction is that the winner is also awarded the rights and obligations accompanying the item, and in particular, the right to listen to the audio compact disk and the obligation not to infringe the copyright of the producers of the music. That is, in reality the auctioneer was not simply selling an audio compact disk, but rather a bundle of goods that include the item itself and certain rights and obligations. We make this transfer of rights and obligations explicit within our framework.

[5] Although the `query` action does not have, as currently implemented in our framework, any physical preconditions, one could imagine scenarios where such an extension would be useful. For example, in a market where an agent's location is important, one might want to restrict agents to only querying attributes of objects that are "close" to the agents, and not be able to query objects that are "further away", even if the agents have the right to do so. Such extensions can be easily accommodated by redefining the query action in the system's libraries.

3.1 Rights

Rights determine the actions that an agent can take, and can arise from ownership or possession of items, or otherwise given to an agent. In their full generality rights are conditional, with their provisions being applicable only under certain conditions.

Definition 8 (Rights) *Let*

```
right(#action, #condition)
```

denote the right to execute action `#action` *whenever condition* `#condition` *is true. The right is* ***exercised*** *by an agent that holds the right if the agent invokes the action* `#action`*.*

As an example, an agent renting a car from 10:00am to 6:00pm might be given a right of the form `right(drive_car, 10:00am` $\leq$ `CurrentTime` $\leq$ `6:00pm)`, where `CurrentTime` denotes the current time of day.[6] If the condition is not met (and given that the agent does not have any other rights on driving the car), then the agent cannot drive the car, by virtue of the restriction principle.

The syntax of conditional rights is sufficiently expressive to account for *perpetual* and *expiring* rights, and for more involved rights, such as the perpetual right to buy bonds, but only once every year and within a limited time span. This last right can be expressed, for instance, as

```
right(buy_bonds, February 1st ≤ CurrentDate ≤ March 1st & not invoked(buy_bonds,
                                                    CurrentYear)),
```

where `CurrentDate` and `CurrentYear` denote, respectively, the current date and year.

It can be useful for an agent to waive its right to perform some action, making in this way a commitment to other agents. In other situations, an agent's rights may be revoked as a result of some violation of that agent with respect to the rules of the market. For instance, an agent failing to pay for an item won at some auction may lose its right to bid in future auctions. Modeling such situations requires agents to have distinct instances of their rights because the destruction of one such instance does not affect the rest of the instances, which can still be exercised. Thus, an agent may have the right to bid on all auctioned items because all agents are given such a right, and may also have the ownership-implied right to bid on any item auctioned by the firm for which the person represented by an agent is an employee. Losing the former right, e.g., because of failed obligations, does not exclude the agent from exercising the latter right in those cases that it applies. This requirement of having multiple instances of rights further motivates our treatment of rights (and obligations) as goods, owned or possessed by agents.

Expiring rights, that is, rights that expire after their use, are also useful. For example, a movie ticket is naturally modeled as a right to see a movie, that expires as soon as it has been exercised. When you exercise the right to see the movie, you simultaneously execute the `see_movie` action and the `give_up(#right)` action; equivalent to your ticket being torn by the person at the theater's door. Note that this is not simply a conditional right, whose condition defines that the `see_movie` action should not have been executed by the agent in order for the right to be applicable. Having seen a movie in the past using a different ticket should not preclude the agent from seeing it again. Expiring rights are handled by the `exercise_once` transaction defined as follows:

[6] For ease of presentation, code fragments presented as part of the discussion will sometimes deviate from the syntax of the scripting language. This is done, for instance, when presenting code fragments dealing with time and the invocation of actions.

```
action(#agent, exercise_once(#right, #action)) :-
    transaction([#action, give_up(#right)]).
```

So, for example, if Bob was to buy a ticket to see a movie, he would have been given a right `#right` of the form `right(exercise_once(#right, see(movie)), true)`, where both appearances of `#right` refer to the name of the same right (which corresponds to the ticket's serial number). Notice that the right is for the `exercise_once` transaction and not the `see(movie)` action, and that the right refers to itself, specifying that it is to be given up once used.

3.2 Obligations

Our framework also adopts obligations, which define sanctions that an agent must face if certain (violating) conditions are met before certain other (satisfying) conditions are met. Without directly constraining an agent's behavior, obligations will indirectly influence the behavior of a rational agent that wishes to avoid sanctions. One can think about obligations as providing *soft* constraints on behavior. In accordance to the free-will principle, an agent freely chooses when and how to satisfy its obligations by appropriately exercising its rights.

Rather than enforcing that agents meet their obligations, for instance by performing planning on an agent's behalf, we let the monitoring system detect violations and impose sanctions, as defined by the domain designer or the participating agents. Sanctions might include the revocation of an agent's rights, the loss of money or possessions, the enforcement of additional obligations, or the banning of an agent from participating in the market mechanism altogether. We will see that it can be useful for an agent to voluntarily adopt an obligation, and (as objects) obligations can also be transferred between agents. The domain designer is responsible for prescribing the class of obligations, which includes the type of their sanctions, that is available to the agents, and the conditions under which these obligations may be taken on or given. Out of the available set of choices, the agents are free to choose how obligations are to be employed, according to their individual goals.

Definition 9 (Obligations) *Let*

```
obligation(#satisfy, #violate, #sanction)
```

denote the obligation of ensuring that condition `#satisfy` *is satisfied no later than condition* `#violate`*, under penalty of invoking action* `#sanction`*.*

Each obligation is associated with two constraints, corresponding, respectively, to the conditions that satisfy or violate the obligation. Since both conditions can be satisfied at the same or different points in the evolution of the state of an economic environment, the one satisfied first determines the status of the obligation. If the two conditions of some obligation are satisfied at the same time, then the obligation is considered to be satisfied. In the case of a violation the appropriate punitive sanction is imposed through the invocation of action `#sanction`. This general form allows us to represent obligations of the following forms:

```
obligation(false, account_balance(alice) < 1000, close_account(alice)),

obligation(account_balance(alice) > 1000, CurrentYear > 2009,
                charge_account(alice, 100)).
```

Assuming that Alice holds such obligations, then in the first case she must ensure that her bank balance does not drop below US $1000 at any time, under penalty of her bank account being closed. In the second case she must ensure that her bank balance goes above US $1000 (but not necessarily stays there) at some time before the end of the year 2009, under penalty of US $100 being deducted from her account.

Agents retain possession of obligations after these expire (violated, or satisfied). This provides an agent with a way to prove that it, or someone else, met or violated an obligation, by allowing other agents to see the status of such obligations through providing appropriate `query` rights.

One question that naturally arises is what happens in the case that the preconditions of the `#sanction` action are not met at the time an agent violates an obligation. As we mentioned, a violated obligation results in the invocation of the `#sanction` action, but not necessarily its execution. We leave to the domain designer and agents that impose and adopt obligations to carefully define meaningful sanctions. As an aid, one can utilize conditional actions, and thus in effect *contingency plans*, with a sanction defined to penalize the violator in a way that makes sense given the current state of the market. Such a sanction, for instance, could be the `charge_or_close_account(alice, 100)` action, whose effects are either to deduct US $100 from Alice's account if the account has at least US $100, or otherwise to close Alice's account.

Sanctions should generally correspond to actions that are undesirable. What is undesirable is, however, highly domain-dependent. Losing ownership of an object or a right might seem to be undesirable but is not always so, since the loss might imply that certain obligations of an agent associated with that object or right are also retracted or satisfied. Consider a company in possession of nuclear waste. A country's laws may impose on every entity the obligation to safely dispose of any nuclear waste they posses. It is evident, then, that losing possession of the nuclear waste, perhaps as an effect of some sanction, is in fact beneficial for the company, since the obligation of proper disposal no longer applies. No universal solution exists as to what constitutes a good sanction for a given domain and part of the burden lies with the domain designer to appropriately define classes of suitable sanctions.

The basic axioms implemented by our framework grant an agent the right to volunteer to take on only certain types of obligations. These are obligations associated with a sanction that is clearly undesirable, namely that of being banned from the market altogether. Depending on the domain description, however, an agent may be granted rights to take on additional types of obligations, and this can provide an opportunity to choose a sanction in a way that it will not be executable when the obligation is violated. This also happens in real markets, as in the case where a company issues a warranty that contains many clauses that effectively prevent the warranty from applying. Other agents are free to judge the weight of the obligation and determine the effective constraints that it implies on the agent's behavior.

3.3 Discussion: A Classification of Obligations in Markets

Before continuing, we find it informative to classify obligations within markets in a hierarchy, based on the power of the obligations. Each level of the hierarchy also naturally corresponds with some level of centralization in the market environment. Roughly, the more powerful the obligation, the more centralization is required to ensure that obligations are met. We consider a hierarchy with four levels and discuss whether each level can be implemented within our framework. The eBay electronic marketplace is adopted as a running example:

Level one: Hard obligations. A hard obligation cannot be violated by an agent and constrains the actions of the agent, forcing it to take specific actions in order to meet an obligation whenever possible. Such hard obligations can only exist when agent actions can be dictated by some centralized entity. We choose not to adopt hard obligations within our framework because they oppose the free-will principle. More than this conceptual disagreement, hard obligations would also require that the monitoring system solve a computationally hard planning problem, to determine hard constraints on the sequences of actions available to each agent. Hard obligations are generally absent from traditional economic environments, although one can argue that the eBay proxy agents provide an example; they act to bid for a good up to some limit while a participant is not winning.

Level two: Soft obligations. A soft obligation is a logical condition associated with punitive sanctions. Such obligations are common in real world markets and legal systems, such as the obligation not to speed associated with the punitive sanction of paying a fine. A centralized entity such as the judicial system of a country monitors for failed obligations and enforces appropriate sanctions. Such a treatment of obligations acknowledges the autonomy of market participants. eBay, for instance, reserves the right to suspend sellers that fail to complete transactions on sold items, but does not *make* a seller complete a transaction. This type of obligation is readily represented in our framework.

Level three: Decentralized enforcement. Obligations with decentralized enforcement do not have a direct consequence. Rather, the violation of some obligation triggers the granting of rights to other participants to take a punitive action against the violator. Whether such actions are actually taken is decided by each of the other participants, based on their own beliefs and goals. In certain U.S. states, one has the government-issued right to shoot a trespasser.[7] On eBay, a participant has an eBay-issued right to leave feedback after completing a transaction. The execution of sanctions of this kind is distributed over the entire population. These obligations are easily represented in our framework, as obligations in which the associated punitive sanction is that of granting rights to other agents.

Level four: Unspecified sanctions. We can also consider obligations that do not have a specified sanction. There is no centralized entity to monitor for violations, and no centralized entity to enforce any sanctions. In these totally decentralized markets, participants can choose to monitor each other for violations, and choose whether (and what) actions are to be taken in response to failed obligations. The sanctions are not prescribed, and are actions that a participant already has the right to execute. Norms in such an environment are emergent, in that no-one specifies or enforces sanctions but certain violated obligations tend to be punished in certain ways. On eBay, buyers might prefer bidding on auctions initiated by individuals as opposed to a large company, essentially punishing a company whose corporate policies they find inappropriate. Consumers choose to leave messages and reviews of hotels on electronic bulletin boards such as `tripadvisor.com`. Such obligations are, of course, easily accommodated within our framework by leaving the sanction empty in an undertaken obligation. The role of the system is to make public the taking on and violation of such obligations and participants are free to choose an appropriate course of action.

[7] The 2005 Florida Statutes, Chapter 776, "Justifiable Use of Force", `http://www.leg.state.fl.us`.

To re-cap, with respect to the hierarchy of obligations considered above, our framework supports obligations at the second level and below. Although we cannot directly handle obligations at the first level (since that would violate the black-box and free-will principles), we can make the sanctions of second-level obligations severe enough via the restriction principle (e.g., ban an agent from participating in the environment altogether) so as to get a similar effect as first-level obligations.

4 Fundamental Axioms

Our framework implements a set of axioms that make precise the concepts of ownership and possession, and also provide a common, shared semantics in defining economic environments, such as the right of an agent to waive its rights or take on certain kinds of obligations.

4.1 Ownership and Possession

Property rights are a basic building block of markets [26, 55] and our framework takes a stand on what the rules governing these rights should look like. To start with, we make an important distinction between ownership and possession. Ownership of an object implies a *bundle of rights*, including the right to use the object and the right to sell it. It also includes the right to sell to other agents various rights to access the object. For possession, we adopt the word *holding*, which we take to mean rightful possession. When one holds something, one has the ability and the right to use the object. However, one *does not have the right to sell it or to sell any rights to it.* This is a common situation in the real world. For example, if Alice rents a car, she has possession of it and the right to use it for a limited time, but she does not have the right to sell it. We make precise the notions of ownership and possession through the following axiomatic definitions.

Axiom 1 (Axiom of Ownership) *We take* ***ownership of a good*** *to be synonymous with owning the right of setting the attributes of the good to values in any manner allowed by the domain actions.*

Our framework implements the Axiom of Ownership by issuing *ownership* of a right of the form

`right(#action, accessible(#action, #agent))`

to every agent `#agent` joining the virtual market, where `accessible(#action, #agent)` is formalized by the monitoring system, and holds exactly when action `#action` only affects attributes of goods owned by agent `#agent`. Thus, the framework formalizes the axiom of ownership in a natural manner, consistent with the treatment followed for general rights. As in the case of other rights, by exercising a right an agent can perform certain action. In the case of the ownership right, by exercising this right the owner of an apple can sell or give possession of the apple, since the effects of these actions are only affecting the `owned_by` and `held_by` attributes of the apple. In the latter case, the agent owning the apple has the right to take the apple back, since the agent still owns the right of setting the possessor of the apple.[8]

Note that the right associated with the Axiom of Ownership is *owned* by agents. This implies that the Axiom of Ownership applies recursively on the associated right itself with the right being

[8] In particular, this implies that an agent owning a right, but not holding it, can still execute an action, since the agent can always reclaim possession of the right, execute the action, and then return the right to its previous possessor, all within a single transaction.

the owned object. Thus, an agent owning a car not only owns the right to drive it, but also owns the right to sell the right to drive the car, to some other agent. Selling the right to use an object without selling the object itself is extremely common in human markets, such as selling someone the right to walk across a piece of land without selling the land. Our treatment of rights as first-class objects allows us to easily express the otherwise involved implications of such natural concepts.

Axiom 2 (Axiom of Possession) *We take* ***possession of a good*** *to imply possession of the right to use the good in a set of prescribed ways associated with the good.*

Our framework implements the Axiom of Possession by issuing *possession* of a right of the form

```
right(#action, (object(#object), value(#object,
[(held_by, #agent), (uses, #uses)]), member(#action, #uses)))
```

to every agent `#agent` joining the virtual market, where `object(#object)` holds exactly when object `#object` exists, and `member` is Prolog's membership testing predicate. The way in which an object may be rightfully used as a consequence of the object's possession is domain-dependent, and can be specified through the attribute `uses`, whose value may be determined at the time of the object's creation by the master agent (according to the domain designer's specifications).

Unlike the Axiom of Ownership, the Axiom of Possession is *held but not owned* by agents. In particular, the agents are not automatically granted the right to sell rights for using objects they possess, even if they themselves possess such rights and are allowed to use those objects.

4.2 Exclusive versus Non-Exclusive Rights

Rights in real life are often not given, but rather issued. When granted the right to walk on a piece of land, the land's owner is still in possession of that same right, because the recipient of the right was not given the owner's instance of the right but was instead issued a copy of the right. This is achieved through the use of an *issuing* action defined by our framework, and references the following axiom:

Axiom 3 (Axiom of Rights) *We take* ***ownership of a right*** `#right` *to imply ownership of the right to issue ownership or possession of right* `#right` *(with non-weaker conditions) to others.*

Our framework implements the Axiom of Rights by issuing *ownership* of a right of the form

```
right(issue_o(right(#action, (#condition, #extra_condition)), #some_agent),
            (object(#right), value(#right, [(owned_by, #agent),
               (instance_of, right(#action, #condition))])))
```

(and a corresponding one where the action `issue_o` of issuing ownership is replaced by the action `issue_p` of issuing possession) to every agent `#agent` joining the virtual market. As a result, if agent `#agent` owns the right `#right` to execute an action `#action` under certain conditions `#conditions`, then agent `#agent` also owns the right to issue ownership (or possession) of the right to execute action `#action` to some agent `#some_agent` given that the same conditions `#conditions`, and possibly additional conditions `#extra_condition`, hold.

As an illustration of the interplay and the recursive nature of the Axiom of Ownership and the Axiom of Rights, our framework supports the following conclusion: if an agent owns a piece of land, then the agent may rightfully grant a second agent possession of the right to sell non-exclusive

rights to third parties for certain uses of the land. The owner of the land need not lose any of its original rights, and in fact may revoke the second agent's right at any point; variations are of course possible depending on the exact right that is granted to the second agent.

4.3 Other Important Axioms

Three additional sets of axioms are defined in our framework. These are implemented in a similar way to the ones described earlier, by issuing ownership of suitable rights to agents when they join the market. The issued rights are presented after the brief description of each set of axioms.

Axioms of Visibility: Agents own the right to query the attributes of all objects that they own, all held rights and obligations, and certain events that occur in a scenario (e.g., the passing of time). These axioms guarantee that objects owned by an agent are exempt from the default treatment of preserving secrecy of objects. The corresponding rights are as follows:

```
right(query(#object, #attribute), (object(#object),
value(#object, [(owned_by, Agent), (#attribute, #value)]))).

right(query(#object, #attribute), (object(#object),
value(#object, [(instance_of, right(#action, #condition)), (held_by,
#agent), (#attribute, #value)]))).

right(query(#object, #attribute), (object(#object),
value(#object, [(instance_of, obligation(#satisfy, #violate, #sanction)),
(held_by, #agent), (#attribute, #value)]))).

right(query(#object, #attribute), (object(#object),
value(#object, [(instance_of, event), (#attribute, #value)]))).
```

In each case, the right to query an attribute `#attribute` of an object `#object` is conditional on the object's existence, and on the object having some value `#value` for that attribute.

Axiom of Commitment: Agents own the right to take on obligations with a sanction of being banned from the market. This provides a minimal means for agents to commit to a particular behavior. The corresponding right is as follows:

```
right(take_on(obligation(#satisfy, #violate, ban(#agent))), true).
```

Of course the domain designer can freely extend the set of obligations an agent can voluntarily take on, according to the domain being modeled. An alternative form of commitment, that of giving up rights that agents own, is also supported and follows from the Axiom of Ownership.

Axiom of Communication: Agents own the right to create `info` objects. The corresponding right is as follows:

```
right(create_info(#info), true).
```

Since created objects are owned by their creator (in this case the agent itself), an agent owns the right to query attributes of the info objects by the Axiom of Visibility, and can issue such a right to other agents by the Axiom of Rights. This guarantees that agents may communicate with each other within the market framework.[9]

A domain designer can further extend the set of axioms for the specific market being modeled by issuing suitable rights to agents upon entry.

5 Implementation

Both the monitoring system and the specification language have been implemented in Prolog and the system is available at `http://www.eecs.harvard.edu/~loizos/norms.html`.

5.1 Prolog, The Standard Library, and Domain Descriptions

Prolog was chosen as an implementation language, recognizing that its goal-oriented computation is a natural fit with the computational tasks of our framework (e.g., checking if conditions are met). The language choice is, of course, accompanied by some compromises on the speed of execution. This concern can be alleviated by implementing computationally-intensive parts of the code using a lower-level language like C, while still maintaining the Prolog code in those parts of the implementation that interface with the domain descriptions.

Before a domain description is loaded by the monitoring system, the framework's *standard library* is loaded. The standard library, as any library in our framework, is simply a domain description, and as such it defines classes, action description laws, and the interventions of the master agent at various occasions. The action description laws implemented by the standard library were presented in Section 2.3.6. The following set of classes are defined by the standard library, and were already discussed in previous sections:

```
event, info, account,
right(#action, #condition),
obligation(#satisfy, #violate, #sanction).
```

Regarding the interventions of the master agent, the standard library does not define the constituent actions of the `initialize` transaction that is invoked by the master agent when initializing the market. Instead, it is left to the domain designer to specify the initial state. On the other hand, for the events involving agents joining or leaving the market, the standard library defines the set of actions that need to be performed by the master agent in order to maintain the market in an economically-sound state. Upon entrance of a new agent, the master agent creates a new account object, initializes its balance to zero, and gives the account to the agent. In addition, the master agent issues a set of rights to the agent as described in Section 4. Upon departure of an agent, the master agent takes ownership/possession of all goods that were previously owned/held by the agent. In particular, if the departing agent holds an object that is owned by another agent, the latter agent retains ownership of the object, and can reclaim possession of the object from the master agent. In an analogous scenario, if the departing agent owns an object that is held by

[9]This does not imply that a domain designer cannot limit what information can be communicated in a specific scenario. Limitations on communication can be captured by imposing obligations on agents not to communicate in prescribed ways.

another agent, the latter agent retains possession of the object. Had these actions not been taken by the master agent, the market would have reached a state where entering agents would not have had their fundamental rights, or where objects would have been owned/held by agents that would have no longer been part of the market; an economically-unsound situation.

A domain description is a Prolog program, and as such it closely follows the syntactic conventions of Prolog. In particular, a domain description may contain the definitions of arbitrary Prolog predicates, which may then be employed in describing the conditions of rights, obligations, actions, or conditional effects. A domain description may import other domain descriptions, or Prolog files in general, in the form of libraries that can be utilized in the former domain description.

The first part of a domain description defines the classes (in addition to the built-in ones) that are to be used, in the manner already described in Section 2.3.1. Each class is assumed to be universally quantified over all its free variables, but the quantification is independent across classes. The next three parts of a domain description define the behavior of the master agent when the market is initialized, and when an agent joins or leaves the market:

```
initially([#action, ..., #action]),
on_entrance(#agent, [#action, ..., #action]),
on_departure(#agent, [#action, ..., #action]).
```

In all cases, `#action` is the name of an action, which the master agent will invoke, following the invocation of the actions specified in the standard library for the corresponding occasions, as discussed earlier. These additional actions may include the creation of new objects and the assigning or taking of objects to or from agents. Each `#action` is assumed to be universally quantified over all its free variables, but independently of the other actions. In the case of the last two constructs, `#agent` is taken to be a universally quantified Prolog variable over the entire construct.

The last and most central part of a domain description defines the actions that are available to agents, as already discussed in Section 2.3.3. As per the Prolog semantics, the free variables in the head of the rules defining the actions are assumed to be universally quantified over the entire action description law. Free variables of the preconditions and effects of an action description rule are each universally quantified over their free variables, but independently of each other. In particular, the effects of an action are decoupled from the action's executability preconditions, with the latter determining only *when*, and not *what* effects are to be produced. State-dependent effects of actions may be represented through conditional effects. Within transactions, each constituent action is universally quantified over its free variables, but independently of each other.

5.2 Communication, History Recording, and Scalability

Communication from agents joining the monitoring system is supported by assigning each agent a private channel, through which all subsequent communication takes place. Thus, each exchanged message is associated with a unique agent, which allows the monitoring system to retain a communication transcript. Communication takes place asynchronously, while the monitoring system employs a continuous treatment of time, with actions occurring instantaneously in the market context. An agent connects to the registration module, e.g., via a public IP address. The registration module instructs the communication module to contact the agent, thus establishing a new private communication channel between the monitoring system and the agent. Through this communication channel the agent can invoke actions.

In a typical execution of the monitoring system, an agent is sent a Prolog list containing all the object attributes of the current state that are visible to the agent. Given the received message, the agent reasons and chooses to invoke some action by replying with the predicate `invoke(#action)`. Messages regarding invoked actions are forwarded through the communication module to the simulation module. The latter handles the execution of the invoked actions, updating the state of the market, and recording the new state in the state database. The simulation module also forwards through the communication module a message to all agents, informing them of the new state of the market (or the part thereof that each agent is allowed to view).

The administrator can communicate with and control the monitoring system through the administrator module. Through that, the administrator can instruct the master agent to execute specific actions (i.e., to intervene in the evolution of the market), or can perform system-related actions, such as closing communication channels, or shutting down the monitoring system. The master agent can be thought to communicate with the monitoring system in the same way as agents do, despite being part of the monitoring system.

As the simulated market passes through different states, the history of states is recorded in a database. This facilitates the generation of a transcript of all activity, including the entrance and departure of agents, the invocations of actions, and the state of the market. This transcript can serve as a proof that the simulation engine is well-functioning, and as a validation of the correctness of any intervening actions performed by the administrator through the master agent. It can also act as a real-life legal contract between agents. For example, if the virtual self of Alice were to sell something to the virtual self of Bob, the transcript could then be thought of as a binding contract between the two to repeat the same transaction in real life. By employing a database, we also reduce the memory overhead in our implementation while incurring only a slight reduction in speed.

Regarding scalability, we are concerned only with execution monitoring, and not planning. For instance, we are not concerned with determining and enforcing a sequence of actions such that an obligation is met. Monitoring remains decidable and tractable as long as the conditions of actions, rights, and obligations are not inherently undecidable or intractable to begin with. Of course, it remains possible that a domain description will lead to an intractable or undecidable computation; this is a result of the fact that the computational power of our framework encompasses the computational power of Prolog. Experimental results using agents and markets we have implemented suggest that such issues are unlikely to arise in natural market descriptions.

6 Example Representations

In this section, we illustrate our framework by providing the representations in our scripting language of four examples: an open outcry English auction, a sealed-bid second-price auction, a combinatorial auction [12, 31], and an exchange following a negotiation process. The representations do not describe the agents participating in the market; the agents can be implemented in some arbitrary language, and their implementation is done outside our framework. Neither do the representations define the process by which the markets clear; the exact process used is chosen and executed by the participating agents. For instance, the winner-determination in a combinatorial auction can be performed using combinatorial optimization, and the agent acting as the auctioneer is responsible for running the appropriate combinatorial optimization algorithm. Rather, the representations define the rules of the markets and capture the important properties for clearing the markets (such as the

fact that the highest bid wins in a case of an auction). The full domain descriptions can be found online, along with the entire system, at `http://www.eecs.harvard.edu/~loizos/norms.html`.

We use boldface to indicate the main language operators and underlining to indicate action names. We substitute certain parentheses with curly brackets to enhance readability. Other than these cosmetic enhancements, we present the domains in the Prolog implementation of the scripting language of our framework. Certain Prolog predicates such as `=` and `\=`, indicating respectively unifiability and non-unifiability, and `;` indicating disjunction, are used throughout.

Objects and actions defined in the standard library are used when necessary. The object `clock` is an instance of the `event` class, and serves as a way to hold the time at which the current state of the world was instantiated. The various predicates used are provided by our framework and were already described in previous sections. The actions `sell(#good,#price,#receiver)` and `ban(#agent)` are imported from the appropriate libraries, with the latter banning agent `#agent` from the market, when executed. The action `issue_p(right(#action,#condition),#agent)` is the built-in action of issuing possession of a right to an agent. When the receiving agent `#agent` is a Prolog variable, the monitoring system interprets the issued right as being held by everyone. We assume, and do not explicitly represent below, the fact that agents have the right to open auctions on items they own. We assume, unless otherwise stated, that bidders have the right to query all the attributes of an auction and all the attributes of the items being auctioned. Such query rights are given to the bidders at the opening of an auction. For ease of exposition, the sanctions in each obligation are defined in terms of the `ban` action.

6.1 Open Outcry English Auction

An open outcry English auction is an ascending-price auction for a single good. We consider an auction with a rolling closure time, so that the auction closes only when there is no more bidding activity. While an auction remains open any bidder can submit ("cry out") a higher bid than the current highest bid. This is the new winning bid price. Upon the auction closing, the good is sold to the highest bidder at the final bid price.[10]

To initialize the auction, an agent owning an item invokes the action of opening an auction. Refer to Figure 2. This provides code for the `open_auction` action, which is defined as a transaction of the `create_auction` action, the taking on of two obligations by the auctioneer agent, and the issuing of rights to the participating agents. The `create_auction` action (not shown here) establishes the auction parameters. The obligations commit the auctioneer to closing the auction and selling the item to the highest bidder soon after that, while the right, given to all bidders, allows them to place bids, conditioned on the new price being higher than the current price.

Bidders proceed to place (and possibly increment) their bid through the `place_bid` action, defined in Figure 3. The `place_bid` action is defined as a transaction of the `raise_bid` action, and an action issuing a right. The `raise_bid` action can only be executed if the agent's bid is higher than the current winning bid and has the effect of updating the state of the auction. The issued right enables the auctioneer to sell the item to the agent based on the bid and consistent with the rules of the auction. At the end of the auction, the auctioneer closes the auction and invokes the `sell` action, as obligated by the rules of the auction.

[10] Bidding the minimal bid increment above the current winning bid while losing and while the bid price is less than an agent's value is an ex post Nash equilibrium of the English auction [31]. This straightforward strategy is a best response whatever the private values of other agents and as long as all agents follow a straightforward strategy. The auction is efficient, in that the item is allocated to the agent with the highest value, in this equilibrium.

```
action(Agent, open_auction(Auction, Item, OpeningPrice)) :-
   transaction([
     create_auction(Auction, Item, OpeningPrice),
     take_on(obligation(                                  % Take on the obligation
       {                                                  % to ensure that
         value(Auction, status, closed)                   % the auction is closed.
       }, {                                               % Satisfy the obligation before
         value(clock, happened_at, Time),                 % the current time exceeds
         value(Auction, last_bid_time, LastBidTime),      % the time of the last placed bid
         atleast(Time, LastBidTime+100)                   % by one hundred time units.
       }, {                                               % Violating the obligation results in
         ban(Agent)                                       % being banned from the market.
       })),
     take_on(obligation(                                  % Take on the obligation
       {                                                  % to ensure that
         value(Auction, [                                 % in the auction
           (status, closed),                              % that has closed
           (highest_bid, HighestBid),                     % with a given highest bid
           (highest_bidder, HighestBidder)                % and a given highest bidder,
         ]),
         HighestBidder = Agent                            % either no bidder placed a bid,
         ; (                                              % or
           object(Event),                                 % a selling action invoked by the
           value(Event, [                                 % auctioneer for the auctioned item,
             (instance_of, event),                        % transferring the item to
             (description, invoked(Agent,                 % the highest bidder and
               sell(Item, HighestBid, HighestBidder),
               successfully)                              % charging the highest bid,
             )                                            % was completed successfully.
           ])
         )
       }, {                                               % Satisfy the obligation before
         value(clock, happened_at, Time),                 % the current time exceeds
         value(Auction, [
           (status, closed),
           (closing_time, ClosingTime)                    % the auction's closing time
         ]),
         atleast(Time, ClosingTime+100),                  % by one hundred time units.
       }, {                                               % Violating the obligation results in
         ban(Agent)                                       % being banned from the market.
       })),
     issue_p(right(                                       % Issue possession of the right
       {                                                  % of
         place_bid(Auction, Bid)                          % placing a bid in the auction,
       }, {                                               % conditioned on
         value(Auction, [                                 % the auction
           (highest_bid, HighestBid),
           (status, open)                                 % being currently open
         ]),                                              % and the bid exceeding by one
         atleast(Bid, HighestBid+1)                       % monetary unit the highest bid.
       }), Bidder)                                        % Everyone can exercise the right.
   ]).
```

Figure 2: The `open_auction` action for an open outcry English auction.

```
action(Agent, place_bid(Auction, Bid)) :-
   transaction([
     raise_bid(Auction, Bid),
     issue_p(right(                                  % Issue possession of the right
       {                                             % of
         sell(Item, Bid, Agent)                      % selling an item to the agent
       }, {                                          % conditioned on
         value(Auction, [                            % the auction
           (status, closed),                         % being currently closed,
           (closing_time, ClosingTime)
           (highest_bid, Bid),                       % the price being the highest bid,
           (highest_bidder, Agent)                   % the agent being the highest bidder
         ]),                                         % and the current time not exceeding
         value(clock, happened_at, Time),            % the auction's closing time
         atleast(ClosingTime+100, Time)              % by one hundred time units.
       }), Auctioneer)                               % Issue the right to an auctioneer.
       where value(Auction, [                        % The issue action is conditional
         (auctioneer, Auctioneer),                   % on the referenced auctioneer and
         (item, Item)                                % item being those of the auction.
     ])
   ]).
```

```
action(Agent, raise_bid(Auction, Bid)) :-
   preconditions([                                   % The action is executable only if
     object(Auction),                                % the agent's bid exceeds
     value(Auction, highest_bid, CurrentBid),        % the current highest bid
     atleast(Bid, CurrentBid+1)                      % by at least one monetary unit.
   ]),
   effects([                                         % The action execution results in
     set(Auction, highest_bid, Bid),                 % the agent's bid being the highest,
     set(Auction, highest_bidder, Agent),            % the agent being the highest bidder,
     set(Auction, last_bid_time, Time)               % and the time of last bid being set
       where value(clock, happened_at, Time)         % to the time of action execution.
   ]).
```

Figure 3: The `place_bid` and `raise_bid` actions for an open outcry English auction.

```
action(Agent, create_auction(Auction, Item, OpeningPrice)) :-
   preconditions([                                          % The action is executable only if
     \+ object(Auction)                                     % the auction does not already exist.
   ]),
   effects([                                                % The action execution results in
     create(Auction, sealed_auction),                       % a sealed-bid auction being created,
     set(Auction, owned_by, Agent),                         % being owned by the agent,
     set(Auction, held_by, Agent),                          % being held by the agent,
     set(Auction, auctioneer, Agent),                       % with the agent as the auctioneer,
     set(Auction, status, open),                            % the auction being currently open,
     set(Auction, item, Item),                              % with the specified auctioned item,
     set(Auction, set_of_bids, [(Agent,OpeningPrice)]),     % an opening price set by the agent,
     set(Auction, winner, undefined),                       % with an undefined winner,
     set(Auction, payment, undefined),                      % an undefined payment,
     set(Auction, closing_time, undefined)                  % and an undefined closing time,
     set(Auction, last_bid_time, Time)                      % and the time of last bid being set
       where value(clock, happened_at, Time),               % to the time of action execution.
   ]).
```

Figure 4: The `create_auction` action in a sealed-bid second-price auction.

6.2 Sealed-Bid Second-Price Auction

In a sealed-bid second-price (Vickrey [60]) auction, each bidder makes a private bid to the auctioneer who commits to sell the item to the highest bidder for the second-highest bid price. Truthful bidding in the Vickrey auction is a dominant strategy equilibrium because the price faced by a bidder is independent of its own bid.[11] As in the case of the open outcry English auction, the auction is initialized upon the `open_auction` action being executed, which along with the obligations taken on by the auctioneer agent, and the rights issued to the participating agents, includes the `create_auction` action (see Figure 4), which establishes the auction parameters. Refer to Figure 5. This provides code for the `open_auction` action. The rights issued during the opening of the auction allow participants to selectively query information about the auction, and to place bids.

One difference in the semantics of this auction from that of the English auction is that the right to query attributes of the auction is conditional on the queried attribute not being the `set_of_bids` attribute. This preserves the privacy of the collected bids. The undertaken obligations ensure that the auction will eventually close, and that the declared winner and payment will be in accordance with the auction's semantics. The auctioneer is also obliged to sell the item to the winner for the specified payment. The predicates `get_first_bidder` and `get_second_price` are implemented by the domain designer using the Prolog syntax and semantics to return the highest bidder and the second highest bid; the implementation of these predicates is straightforward.

Refer to Figure 6. Bidders proceed to submit sealed bids, by updating the `set_of_bids` attribute, but without ever seeing its actual contents. Each bidder only has a right to place one bid. Finally, the auctioneer closes the auction by declaring a winner and a payment and invokes the appropriate `sell` action.

[11] The Vickrey auction is efficient, allocating the good to the agent with the highest value, in private value environments. Furthermore, when the Vickrey auction is coupled with a reservation price (so that the item is not sold when the highest bid is below some value), then it is *revenue-maximizing* with identically, independently distributed values [39].

```
action(Agent, open_auction(Auction, Item, OpeningPrice)) :-
   transaction([
     create_auction(Auction, Item, OpeningPrice),
     take_on(obligation(                                  % Take on the obligation to
       {                                                  % ensure that
         value(Auction, [                                 % the auction
           (status, closed),                              % is closed,
           (set_of_bids, SetOfBids),                      % and that given the placed bids,
           (winner, HighestBidder),                       % the determined winner
           (payment, SecondHighestBid)                    % and corresponding payment
         ]),                                              % are respectively
         get_first_bidder(SetOfBids, HighestBidder)       % the highest bidder
         get_second_price(SetOfBids, SecondHighestBid),   % and the second highest bid.
       }, {                                               % Satisfy the obligation before
         value(clock, happened_at, Time),                 % the current time exceeds
         value(Auction, last_bid_time, LastBidTime),      % the time of the last placed bid
         atleast(Time, LastBidTime+100)                   % by one hundred time units.
       }, {                                               % Violating the obligation results in
         ban(Agent)                                       % being banned from the market.
       })),
     take_on(obligation(                                  % Take on the obligation to
       {                                                  % ensure that
         value(Auction, [                                 % in the auction
           (status, closed),                              % that has closed
           (winner, HighestBidder),                       % with a given winner
           (payment, SecondHighestBid)                    % and a given payment,
         ]),
         HighestBidder = Agent                            % either no bidder placed a bid,
         ; (                                              % or
           object(Event),                                 % a selling action invoked by the
           value(Event, [                                 % auctioneer for the auctioned item,
             (instance_of, event),                        % transferring the item to
             (description, invoked(Agent,                 % the highest bidder and
               sell(Item, SecondHighestBid, HighestBidder),
               successfully)                              % charging the second highest bid,
             )                                            % was completed successfully.
           ])
         )
       }, {                                               % Satisfy the obligation before
         value(clock, happened_at, Time),                 % the current time exceeds
         value(Auction, [
           (status, closed),
           (closing_time, ClosingTime)                    % the auction's closing time
         ]),
         atleast(Time, ClosingTime+100),                  % by one hundred time units.
       }, {                                               % Violating the obligation results in
         ban(Agent)                                       % being banned from the market.
       })),
     issue_p(right(                                       % Issue possession of the right
       {                                                  % of
         query(Auction, QueriedAttribute)                 % querying an auction attribute
       }, {                                               % conditioned on
         value(Auction, status, open),                    % the auction being currently open
         QueriedAttribute \= set_of_bids                  % and not querying the placed bids.
       }), Bidder),                                       % Everyone can exercise the right.
     issue_p(right(                                       % Issue possession of the right
       {                                                  % of
         place_bid(Auction, Bid)                          % placing a bid in the auction,
       }, {                                               % conditioned on
         value(Auction, status, open)                     % the auction being currently open.
       }), Bidder)                                        % Everyone can exercise the right.
   ]).
```


Figure 5: The `open_auction` action in a sealed-bid second-price auction.

```
action(Agent, place_bid(Auction, Bid)) :-
   transaction([
     submit_bid(Auction, Bid),
     issue_p(right(                                  % Issue possession of the right
       {                                             % of
         sell(Item, Bid, Agent)                      % selling an item to the agent
       }, {                                          % conditioned on
         value(Auction, [                            % the auction
           (status, closed),                         % being currently closed,
           (closing_time, ClosingTime),              % and the auction determining
           (winner, Agent),                          % the agent as the winner and the
           (payment, Bid)                            % payment as the auction payment,
         ]),                                         % and the current time not exceeding
         value(clock, happened_at, Time),            % the auction's closing time
         atleast(ClosingTime+100, Time)              % by one hundred time units.
       }), Auctioneer)                               % Issue the right to an auctioneer.
       where value(Auction, [                        % The issue action is conditional
         (auctioneer, Auctioneer),                   % on the referenced auctioneer and
         (item, Item)                                % item being those of the auction.
       ])
   ]).
```

```
action(Agent, submit_bid(Auction, Bid)) :-
   preconditions([                                   % The action is executable only if
     object(Auction),                                % the auction exists and
     value(Auction, set_of_bids, SetOfBids),         % the currently placed bids do
     \+ member((Agent,AnyBid), SetOfBids)            % not include a bid by the agent.
   ]),
   effects([                                         % The action execution results in
     set(Auction, set_of_bids, [(Agent,Bid)|SetOfBids])  % the new bid being added
       where value(Auction, set_of_bids, SetOfBids),   % in the set of existing bids,
     set(Auction, last_bid_time, Time)               % and the time of last bid being set
       where value(clock, happened_at, Time)         % to the time of action execution.
   ]).
```

Figure 6: The `place_bid` and `submit_bid` actions in a sealed-bid second-price auction.

6.3 Sealed-Bid Combinatorial Auction

Our third example is that of a sealed-bid combinatorial auction (CA). In a combinatorial auction [47], there are multiple, distinct goods to auction and each agent may have substitutes ("I want only A or B.") or complements ("I want only A and B.") valuations. To keep things simple, we consider the case of valuations that are described in an *exclusive-or* (XOR) bidding language [41]. Each agent can make multiple bids, with each bid specifying its maximum willingness-to-pay for a distinct bundle of items. The XOR language semantics means that only one of these bids can be accepted. Each agent can submit any number of such bids and it is easy to see that the XOR language is expressive (if not necessarily concise), since an agent can specify an explicit value for every possible bundle.

The particular CA illustrated here is a generalized Vickrey auction, which instantiates the Vickrey-Clarke-Groves (VCG) mechanism to the CA domain, and generalizes the sealed-bid second-price auction described in the previous section. Each agent makes a payment equal to the marginal externality that it imposes on the rest of the system. In the special case of a single item auction this is exactly the second-price auction; the winner's payment is the second-highest bid, which is the value that would have been achieved without the presence of the winner. See Krishna [31] and Cramton et al. [12] for details.

We include this example because it illustrates the manner with which NP-hard optimization problems can be incorporated into our framework. The winner determination problem in CAs with XOR bids, which is to find a set of disjoint bids that maximize the total revenue, is NP-hard [32, 47]. But very large instances can be solved in practice [49] and we can handle this by allowing the auctioneer, which is implemented outside of the framework, to use state-of-the-art algorithms (such as branch-and-bound with linear programming (LP) heuristics) to solve the problem and provide a proof of the optimality of its solution back to the monitoring system.

The auctioneer takes on an obligation to generate a proof to establish that the allocation is correct. The Prolog-based monitoring system can establish correctness, but is freed from the burden of solving the winner-determination problem (which is contained to the auctioneer). Refer to Figure 7. This illustrates a snippet of the `open_auction` action for a sealed-bid CA. We focus on the place where this differs from the sealed-bid second-price single-item auction, providing the obligation adopted by the auctioneer agent that relates to the correctness of the outcome of the auction. Because the scripting language is built on top of Prolog, the guarantees can be expressed as arbitrary Prolog predicates; our monitoring system then verifies that the predicates hold in a particular run of the market. On closing the auction, the auctioneer determines the revenue-maximizing allocation and the VCG payments by solving a sequence of optimization problems. First the optimal allocation `Allocation` is determined, and subsequently the optimal allocations `AllocationPerMarginalMarket` in each of the marginal economies, in which each agent is removed from the optimization problem in turn (see Krishna [31]). The optimality of the solution to each of these problems is verified.

In generating a proof of correctness for the optimality of an allocation the auctioneer can adopt one of two approaches. One approach is based on competitive equilibrium (CE) prices. Prices are CE when the allocation has the property that: *(i)* every bidder maximizes its utility (value-price) at the prices with the bundle that it is allocated; *(ii)* the seller maximizes its revenue with the allocation across all feasible allocations [5]. Property *(i)* is easy to verify when each bidder submits only a small number of XOR bids, by a linear scan of the bids of each bidder to check

```
action(Agent, open_auction(Auction, Item, OpeningPrice)) :-
   transaction([
     ...
     take_on(obligation(                                    % Take on the obligation
       {                                                    % to ensure that
         value(Auction, [                                   % the auction is closed,
           (status, closed),                                % and that given the placed bids,
           (set_of_bids, SetOfBids),                        % the determined payments,
           (payments, Payments),                            % actual allocation / prices,
           (allocation, Allocation),                        % and marginal allocations / prices
           (prices, Prices),
           (marginal_allocations, AllocationPerMarginalMarket),
           (marginal_prices, PricesPerMarginalMarket)
         ]),
         AllAllocations =                                   % the allocation is efficient
           [Allocation|AllocationPerMarginalMarket],        % and the determined payments are
         AllPrices =                                        % the VCG payments of the market.
           [Prices|PricesPerMarginalMarket],
         checkOutcomeEfficiency(SetOfBids, AllAllocations, AllPrices),
         checkVCGPayments(SetOfBids, AllAllocations, Payments)
       }, {                                                 % Satisfy the obligation before
         value(clock, happened_at, Time),                   % the actual time exceeds
         value(Auction, last_bid_time, LastBidTime),        % the time of the last placed bid
         atleast(Time, LastBidTime+100)                     % by one hundred time units.
       }, {                                                 % Violating the obligation results in
         ban(Agent)                                         % being banned from the market.
       })),
   ]).
```

Figure 7: The `open_auction` action for a sealed-bid, generalized Vickrey (combinatorial) auction.

that it is allocated the bundle that maximizes its utility.[12] Moreover, when there are simple CE prices, with the prices quoted on items and the price on a bundle defined as the sum of the price on the items in the bundle, then Property *(ii)* can be verified by simply checking that every item with a non-zero price is allocated. This is the approach that is assumed in Figure 7, in which the `checkOutcomeEfficiency` predicate receives the sequence of allocations and prices `AllPrices` that correspond to each allocation. On the other hand, when such simple prices exist then the winner determination problem is in `P` because it can be solved via a LP. We provide a second, generally applicable approach, next.

In the general, more interesting case, we suggest an algorithm-specific method to verify Property *(ii)*. Suppose that the auctioneer adopts a systematic, branch-and-bound [40] algorithm to find the optimal allocation. The proof provided to the monitoring system would consist of the state of the final branch-and-bound tree. Namely, the auctioneer would provide the branching decision that was made at every internal node, the evaluation of the allocations on the leaves, the linear programming bound for each part of the search tree that is fathomed, and a proof of this bound.[13] Happily, where a proof of an LP bound is required this can be provided by simply reporting the feasible primal and dual solution to the LP relaxation at that node. If the LP bound is correct then the feasible primal and dual solutions will both adopt the value of this bound. Both primal and dual solutions can be concisely described, and can be checked quickly (in time linear in the number of items, number of bids, and number of agents) by the monitoring system.

Once the correctness of each allocation is established then the VCG payments are verified with a call to the `checkVCGPayments` predicate. This step is trivial and proceeds by simple numerical evaluation of each allocation given the bids. Both the `checkOutcomeEfficiency` and `checkVCGPayments` predicates are implemented in Prolog and are part of the domain description provided by the domain designer. The definition of these predicates, along with the entire domain description, can be made available for inspection by interested parties so that their correct specification may be verified.

6.4 Exchange Following Negotiation

Unlike the auction examples that have been presented above where multiple agents participate and compete against each other to win a (set of) good(s), we now consider an exchange, where two agents negotiate on what goods to exchange between them. We consider a negotiation process where the participants take turns in offering some good. At any point the agent whose turn is to make an offer may choose to leave the negotiation process. After each of the participants has made at least one offer, negotiation may continue, or the agent whose turn is to make an offer may choose to accept the current exchange. At this point the exchange happens, and the process terminates.

Although a negotiation process involves only two participants, a third agent plays the role of the coordinator, much in the same way that an auctioneer coordinates an auction. Although it is possible for one of the two participants to act as the coordinator, it is conceptually cleaner to have a third party play this role. We adopt this view here. The coordinator invokes the action of opening a negotiation. Refer to Figure 8. This provides code for the `open_exchange` action,

[12] Note that the winner determination problem remains `NP`-hard even if each bidder submits only one bundle. This is the so-called "single minded combinatorial auction" problem [33].

[13] In LP-based branch and bound, one bounds the value of entire subtrees by solving the linear-programming relaxation at a node. If this is smaller than the best, feasible allocation found so far, then no further (enumerative) search is required below that node.

```
action(Agent, open_exchange(Exchange, Party1, Party2)) :-
   transaction([
     create_exchange(Exchange, Party1, Party2),
     issue_p(right(                                    % Issue possession of the right
       {                                               % of
         make_offer(Exchange, Item)                    % offering an item in the exchange,
       }, {                                            % conditioned on
         value(Exchange, [
           (turn, Party),                              % being the party's turn
           (status, Status)                            % and the status of the exchange
         ]),
         member(Status, [initial, pending])            % allowing offers to be made.
       }), Party)                                      % Issue the right to
       where member(Party, [Party1, Party2])           % both participating parties.
     issue_p(right(                                    % Issue possession of the right
       {                                               % of
         accept_offer(Exchange)                        % accepting the offer in the exchange,
       }, {                                            % conditioned on
         value(Exchange, [
           (turn, Party),                              % being the party's turn
           (status, pending)                           % and the status of the exchange
         ])                                            % allowing offers to be accepted.
       }), Party)                                      % Issue the right to
       where member(Party, [Party1, Party2])           % both participating parties.
     issue_p(right(                                    % Issue possession of the right
       {                                               % of
         reject_offer(Exchange)                        % rejecting the offer in the exchange,
       }, {                                            % conditioned on
         value(Exchange, [
           (turn, Party),                              % being the party's turn
           (status, Status)                            % and the status of the exchange
         ]),
         member(Status, [initial, pending])            % allowing offers to be rejected.
       }), Party)                                      % Issue the right to
       where member(Party, [Party1, Party2])           % both participating parties.
   ]).
```

Figure 8: The `open_exchange` action for an exchange following negotiation.

which is defined as a transaction of the `create_exchange` action, and the issuing of rights to the participating agents. The `create_exchange` action (not shown here) establishes the negotiation parameters: essentially slots for information to be stored during the negotiation process. The rights allow the participants to update their offer, accept the current offer, or leave the negotiation.

The participants proceed to make (and possibly update) their offer through the `make_offer` action, defined in Figure 9. The `make_offer` action is defined as a transaction of the `update_offer` action, and an action of taking on an obligation. The `update_offer` action has the effect of updating the good offered by the participant making an offer, and making it the turn of the other agent to make an offer. The obligation commits the participant making the offer to give ownership of the good to the other participant if the offer is accepted.

The action for accepting the current offer simply changes the status of the exchange to indicate the acceptance of the offer, thus precluding any additional offers, and initiating a time limit for exchange of the goods. Similarly, the action for leaving an exchange simply changes the status of

```
action(Agent, make_offer(Exchange, Item)) :-
   transaction([
     update_offer(Exchange, Item),
     take_on(obligation(                                  % Take on the obligation to
       {                                                  % ensure that
         value(Exchange, offers, Offers),                 % either the agent
         \+ member((Agent, Item), Offers)                 % updates its offer,
         ; (                                              % or
           member((OtherParty, OtherItem), Offers),       % the other party
           OtherParty \= Agent,                           % participating in the exchange
           value(Item, owned_by, OtherParty)              % owns the item offered by the agent.
         )
       }, {                                               % Satisfy the obligation before
         value(clock, happened_at, Time),                 % the current time exceeds
         value(Exchange, [
           (status, accepted),
           (closing_time, ClosingTime)                    % the negotiation's closing time
         ]),
         atleast(Time, ClosingTime+100),                  % by one hundred time units.
       }, {                                               % Violating the obligation results in
         ban(Agent)                                       % being banned from the market.
       })),
   ]).
```

Figure 9: The `make_offer` action for an exchange following negotiation.

the exchange to indicate that the exchange is closed.

7 Conclusions

Rights and obligations, important in human economies and often enforced through legal remedies, will be important in agent-mediated economies in providing well-defined semantics and in enabling the construction of useful economic mechanisms. We have defined a formal language that allows the specification of economic environments and is paired with a monitoring system that allows for the automatic checking of rights and the enforcement of sanctions based on failed obligations. Central to our approach is the novel treatment of rights and obligations as first-class goods.

Building on well-defined design principles, which are demonstrably consistent with existing electronic markets such as eBay, our framework facilitates the representation of complex markets, and allows for a natural treatment of central concepts in economic markets, such as those of ownership and possession. To the best of our knowledge this work is the first to represent these notions in a completely formal language, which at the same time allows the specification of the dynamic nature of economic markets in which these notions gain value through their utilization by rational agents that pursue their goals. The complex and recursive rights that follow from ownership and possession are straightforwardly represented in our framework, capitalizing on our treatment of rights (and obligations) as goods themselves, on which their provisions can recursively apply. Our framework also offers a rich set of action constructs, allowing conditional, non-deterministic, and even spontaneous changes in the states of markets (as a result of events exogenous to the market participants), as well as transactions, and contingency plans.

It is our hope that in exposing the semantics of markets to automated agents, this framework

will promote further research into the agent-based reasoning within electronic markets, for instance by enabling simulation platforms for testing agent designs. Orthogonally, we also hope that this work will promote further efforts to formalize, in a computer-compliant manner, other notions often encountered in economic markets, leading in turn to the automation and agent-mediation of markets that rely today on human involvement.

Acknowledgments

Useful comments and suggestions were received during seminar presentations of earlier versions of this work in the Harvard School of Engineering and Applied Sciences, and the Workshop on Agent Mediated Electronic Commerce VI. Thanks to Adam Wyner for helpful comments and guidance. Thanks also to the anonymous reviewers for helpful, constructive comments received on an earlier draft of this paper. This work was supported in part by NSF grants IIS-0238147 and IIS-0091815.

References

[1] H. M. Aldewereld, A. Garcia-Camino, F. P. M. Dignum, P. Noriega, J. A. Rodriguez-Aguilar, and C. Sierra. Operationalisation of norms for usage in electronic institutions. In *Proceedings of Fifth International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'06)*, pages 223–225, May 2006.

[2] H. M. Aldewereld, J. Vázquez-Salceda, F. Dignum, and J.-J. Ch. Meyer. Verifying norm compliance of protocols. In O. Bossier, J. Padget, V. Dignum, G. Lindeman, E. Matson, S. Ossowski, J. Sichman, and J. Vázquez-Salceda, editors, *Coordination, Organisation, Institutions and Norms in Agent Systems I*, pages 222–236. Berlin: Springer-Verlag, 2006.

[3] J. L. Arcos, M. Esteva, P. Noriega, J. A. Rodriguez-Aguilar, and C. Sierra. Engineering open environments with electronic institutions. *Journal on Engineering Applications of Artificial Intelligence*, 18(2):191–204, March 2005.

[4] A. Artikis, J. Pitt, and M. Sergot. Animated specifications of computational societies. In *Proceedings of First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'02)*, pages 1053–1061, July 2002.

[5] S. Bikhchandani and J. M. Ostroy. The package assignment model. *Journal of Economic Theory*, 107(2):377–406, December 2002.

[6] G. Boella and L. van der Torre. Contracts as legal institutions in organizations of autonomous agents. In V. Dignum, D. Corkill, C. Jonker, and F. Dignum, editors, *Proceedings of Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'04)*, pages 948–955, August 2004.

[7] G. Boella and L. van der Torre. Norm negotiation in multiagent systems. *International Journal of Cooperative Information Systems*, 2006.

[8] M. Boman. Norms in artificial decision making. *Artificial Intelligence and Law*, 7:17–35, 1999.

[9] R. I. Brafman and M. Tennenholtz. On partially controlled multi-agent systems. *Journal of Artificial Intelligence Research*, 4:477–507, 1996.

[10] J. Carmo and A. Jones. Deontic logic and contrary-to-duties. In *Handbook of Philosophical Logic*, volume 3, pages 203–279. Kluwer, 2001.

[11] C. Castelfranchi, R. Conte, and M. Paolucci. Normative reputation and the cost of compliance. *Journal of Artificial Societies and Social Simulation*, 1, 1998.

[12] P. Cramton, Y. Shoham, and R. Steinberg, editors. *Combinatorial Auctions*. MIT Press, January 2006.

[13] G. Cuni, M. Esteva, P. Garcia, E. Puertas, C. Sierra, and T. Solchaga. MASFIT: Multi-agent systems for fish trading. In *Proceedings of Sixteenth European Conference on Artificial Intelligence (ECAI'04)*, pages 710–714, August 2004.

[14] P. d'Altan, J.-J. Ch. Meyer, and M. Wieringa. An integrated framework for ought-to-be and ought-to-do constraints. *Artificial Intelligence and Law*, 4:77–111, 1996.

[15] A. Daskalopulu and T. S. E. Maibaum. Towards electronic contract performance. In *Proceedings of International Workshop on Legal Information Systems and Applications (LISA'01)*, pages 771–777, 2001.

[16] C. Dellarocas, M. Klein, and J. A. Rodriguez-Aguilar. An exception-handling architecture for open electronic marketplaces of ContractNet software agents. In *Proceedings of Second ACM Conference on Electronic Commerce (EC'00)*, pages 225–232, October 2000.

[17] M. Estava. *Electronic Institutions: From Specification to Development*. PhD thesis, Politechnica de Catalunya, 2003.

[18] M. Esteva, D. de la Cruz, and C. Sierra. ISLANDER: An electronic institutions editor. In *Proceedings of First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'02)*, pages 1045–1052, July 2002.

[19] M. Esteva, J. A. Rodriguez-Aguilar, C. Sierra, P. Garcia, and J. L. Arcos. On the formal specifications of electronic institutions. In F. Dignum and C. Sierra, editors, *Agent-Mediated Electronic Commerce: The European AgentLink Perspective*, volume LNCS 1991, pages 126–147. Spring-Verlag, 2001.

[20] N. Fornara and M. Colombetti. Specifying and enforcing norms in artificial institutions. In A. Omicini, B. Dunin-Keplicz, and J. Padget, editors, *Proceedings of Fourth European Workshop on Multi-Agent Systems (EUMAS'06)*, 2006.

[21] A. Garcia-Camino, P. Noriega, and J. A. Rodriguez-Aguilar. Implementing norms in electronic institutions. In *Proceedings of Fourth International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'05)*, pages 667–673, July 2005.

[22] D. Grossi, H. Aldewereld, and F. Dignum. Ubi lex, ibi poena: Designing norm enforcement in e-institutions. In V. Dignum, N. Fornara, and P. Noriega, editors, *Proceedings of AAMAS'06 Workshop on Coordination, Organization, Institutions and Norms in Agent Systems (COIN'06)*, pages 107–120, May 2006.

[23] R. Guttman, A. Moukas, and P. Maes. Agent-mediated electronic commerce: A survey. *Knowledge Engineering Review*, 13(2):147–159, July 1998.

[24] C. Hanachi and C. Sibertin-Blanc. Protocol moderators as active middle-agents in multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 8(2):131–164, March 2004.

[25] H. L. A. Hart. *The Concept of Law*. Clarendon Press, Oxford, 1961.

[26] O. D. Hart. *Firms, Contracts, and Financial Structure*. Oxford University Press, 1995.

[27] M. He, N. R. Jennings, and H. Leung. On agent-mediated electronic commerce. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):985–1003, July/August 2003.

[28] M. O. Jackson. Mechanism theory. In *The Encyclopedia of Life Support Systems*. EOLSS Publishers, 2000.

[29] S. L. Peyton Jones and J-M. Eber. How to write a financial contract. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*. Palgrave Macmillan, 2003.

[30] J. Kephart and A. R. Greenwald. Shopbot economics. *Autonomous Agents and Multi-Agent Systems*, 5(3):255–287, September 2002.

[31] V. Krishna. *Auction Theory*. Academic Press, 2002.

[32] D. Lehmann, R. Muller, and T. W. Sandholm. The winner determination problem. In Cramton et al. [12], chapter 12.

[33] D. Lehmann, L. I. O'Callaghan, and Y. Shoham. Truth revelation in approximately efficient combinatorial auctions. *Journal of the ACM*, 49(5):577–602, September 2002.

[34] A. Lomuscio and M. Sergot. Deontic interpreted systems. *Studia Logica*, 75:63–92, 2003.

[35] F. López y Lopez, M. Luck, and M. d'Inverno. A normative framework for agent-based systems. *Computational and Mathematical Organization Theory*, 12:227–250, 2006.

[36] P. Maes, R. Guttman, and A. Moukas. Agents that buy and sell: Transforming commerce as we know it. *Communications of ACM*, 42(3):81–91, March 1999.

[37] L. T. McCarty. Permissions and obligations. In *Proceedings of Tenth International Joint Conference on Artificial Intelligence (IJCAI'83)*, pages 287–294, August 1983.

[38] J.-J. Ch. Meyer and R. J. Wieringa, editors. *Deontic Logic in Computer Science: Normative System Specification*. John Wiley & Sons, 1994.

[39] R. B. Myerson. Optimal auction design. *Mathematics of Operation Research*, 6(1):58–73, February 1981.

[40] G. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization*. Wiley-Interscience, 1999.

[41] N. Nisan. Bidding languages for combinatorial auctions. In Cramton et al. [12], chapter 9.

[42] A. Omicini, A. Ricci, M. Viroli, C. Castelfranchi, and L. Tummolini. Coordination artifacts: Environment-based coordination for intelligent agents. In *Proceedings of Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'04)*, pages 286–293, August 2004.

[43] J. Padget and R. Bradford. A π-calculus model of a Spanish fish market. In *Proceedings of First International Workshop on Agent Mediated Electronic Trading (AMET'98)*, pages 166–188, May 1998.

[44] J. A. Rodriguez-Aguilar. *On the Design and Construction of Agent-Mediated Electronic Institutions*. PhD thesis, IIIA, 2001.

[45] J. A. Rodriguez-Aguilar, P. Noriega, C. Sierra, and J. Padget. FM96.5 A Java-based electronic auction house. In *Proceedings of Second International Conference on The Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'97)*, pages 207–224, April 1997.

[46] J. S. Rosenschein and G. Zlotkin. Designing conventions for automated negotiation. *AI Magazine*, 15(3):29–46, Fall 1994.

[47] M. H. Rothkopf, A. Pekeč, and R. M. Harstad. Computationally manageable combinatorial auctions. *Management Science*, 44(8):1131–1147, 1998.

[48] T. W. Sandholm. Automated negotiation. *Communications of the ACM*, 42(3):84–85, March 1999.

[49] T. W. Sandholm. Optimal winner determination algorithms. In Cramton et al. [12], chapter 14.

[50] C. Sierra. Agent-mediated electronic commerce. *Autonomous Agents and Multi-Agent Systems*, 9(3):285–301, November 2004.

[51] C. Sierra and P. Noriega. Agent-mediated interaction. From auctions to negotiation and argumentation. In M. d'Inverno, M. Luck, M. Fisher, and C. Preist, editors, *Lecture Notes in Artificial Intelligence*, volume 2403, pages 27–48. Springer Verlag, May 2002.

[52] P. Stone and A. R. Greenwald. The First International Trading Agent Competition: Autonomous Bidding Agents. *Electronic Commerce Research*, 5(2):229–265, April 2005.

[53] Y-H. Tan and W. Thoen. A logical model of directed obligations and permissions to support electronic contracting. *International Journal of Electronic Commerce*, 3(2):87–104, December 1998.

[54] M. Tennenholtz. On stable social laws and qualitative equilibria. *Artificial Intelligence*, 102:1–20, 1998.

[55] J. Tirole. Incomplete contracts: Where do we stand? *Econometrica*, 67(4):741–781, July 1999.

[56] L. van der Torre, J. Hulstijn, M. Dastani, and J. Broersen. Specifying multiagent organizations. In A. Lomuscio and D. Nute, editors, *Deontic Logic in Computer Science (DEON'04)*. Berlin: Springer-Verlag, 2004.

[57] J. Vázquez-Salceda, H. Aldewereld, and F. Dignum. Norms in multiagent systems: From theory to practice. *International Journal of Computer Systems Science and Engineering*, 20:225–236, 2005.

[58] J. Vázquez-Salceda and F. Dignum. Modeling electronic organizations. In *Multi-Agent Systems and Applications III*, volume LNAI 2691, pages 584–593. Springer Verlag, 2003.

[59] J. Vázquez-Salceda, V. Dignum, and F. Dignum. Organizing multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 11(3):307–360, November 2005.

[60] W. Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *Journal of Finance*, 16(1):8–37, March 1961.

[61] M. P. Wellman, A. R. Greenwald, and P. Stone. *Autonomous Bidding Agents: Strategies and Lessons from the Trading Agent Competition*. MIT Press, 2007.

[62] R. J. Wieringa and J.-J. Ch. Meyer. Applications of deontic logic in computer science: A concise overview. In *Deontic Logic in Computer Science: Normative System Specification*, pages 17–40. John Wiley and Sons Ltd., 1994.

[63] H. C. Wong and K. Sycara. A taxonomy of middle-agents for the Internet. In *Proceedings of Fourth International Conference on Multi-Agent Systems (ICMAS'00)*, pages 465–466, July 2000.

[64] M. Wooldridge, N. R. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, September 2000.

[65] P. R. Wurman, M. P. Wellman, and W. E. Walsh. Specifying rules for electronic auctions. *AI Magazine*, 23(3):15–23, Fall 2002.

[66] A. Z. Wyner. Maintaining obligations on stative expressions in a deontic action logic. In A. Lomuscio and D. Nute, editors, *Deontic Logic in Computer Science (DEON'04)*, pages 258–274. Berlin: Springer-Verlag, 2004.

[67] A. Z. Wyner. A functional program for agents, actions, and deontic specifications. In *Proceedings of Fourth International Workshop on Declarative Agent Languages and Technologies*, 2006.

[68] A. Z. Wyner. Sequences, obligations and the contrary-to-duty paradox. In *Proceedings of Eighth International Workshop on application of Deontic Logic to Computer Science (DEON'06)*, volume 4048. Springer Lecture Notes in AI, 2006.