Applying Dependency Injection to Agent-Based Modeling: the JABM Toolkit

Steve Phelps

Centre for Computational Finance and Economic Agents (CCFEA)

sphelps@essex.ac.uk

August 31, 2012

Abstract

In many areas of science, agent-based models have become increasingly important. These models are often sufficiently complex that deriving closed-form solutions for quantitative aspects of their macroscopic behaviour is often impractical if not impossible. Thus these models are often analysed using Monte-Carlo simulation. Object-oriented programming languages are naturally suited to implementing such models. However, Monte-Carlo methods impose some subtle requirements: we must run the same program very many times with randomly-drawn values of variables in the simulation being drawn from probability distributions, taking great care to ensure that each run is not contaminated with state from previous runs. Typically these problems are tackled by model-specific application code residing in the objects representing individuals in the model. However, this approach is not declarative and leads to cross-contamination between code representing experiments or treatments and code representing the model itself. In this paper we give an overview of agent-based modelling with an emphasis on its application to multi-agent systems research, and show how a design pattern called "dependency injection" can be used to implement highly configurable simulation models which are able to incorporate various assumptions about agents' rationality and learning.

1 Introduction

In this paper we describe practical aspects of developing agent-based simulation models with particular emphasis on agent-based computational economics and work carried out within the trading-agent design

and analysis community. We introduce a new software framework for building agent-based models — the Java Agent Based Modeling (JABM) toolkit[1]. Work on JABM arose out of an earlier project — Java Auction Simulator API (JASA)[2] — which was designed as a toolkit for performing experiments in Agent-based Computational Economics (ACE)[3] and was subsequently used as the basis of the JCAT software used to run the CAT tournament which is part of the Trading Agent Competition (Cai et al, 2009).

JABM is an attempt to build a library of software components that can be used for more general agent-based models, and now serves as the underlying framework used in the implementation of JASA. One of the challenges in building this framework is that, even within the trading-agent research community, there are many different approaches to modelling and analysing agents' decision making; for example all of the following approaches are currently used: (i) game-theoretic approximation based on the empirical game-theory methodology (Wellman, 2006), (ii) individual and multi-agent reinforcement learning, and (iii) adaptation through (co)-evolutionary processes. A unique aspect of JABM is that it synthesises all of these approaches under a common framework, allowing agent-based models that are developed using JABM to be analysed under a wide variety of assumptions regarding agent rationality.

JABM attempts to uphold the principle that entities in our simulation model should be represented using objects, and avoids introducing additional language constructs over and above those that are naturally available in Object-Oriented (OO) languages such as Java. In the jargon of modern software engineering, where possible we prefer to use Plain Old Java Objects (POJOs)[4] to represent agents, events and any other entities in our simulation model. Moreover, because of its object-oriented design, simulations in JABM can be reconfigured to use different learning mechanisms without rewriting any of the code representing the underlying model or the agents themselves. This design philosophy is also applied to other aspects of the framework, particularly to the modelling of free parameters; JABM uses an object-oriented design pattern called dependency-injection (Fowler, 2004) to allow the underlying model to be executed very many times with different randomly-drawn values for free parameters. By using dependency-injection we are able to express the configuration of the model, including its statistical properties, declaratively. This allows us to run the same model under different assumptions regarding, e.g. initial conditions and free parameter distributions without making any modifications to the code representing the model itself.

In this paper we give an overview of the JABM framework and how it has been applied to several areas of research in the field of multi-agent systems. The remainder of the paper is outlined as follows. In the

[1] Phelps (2011a)
[2] Phelps (2011b)
[3] Tesfatsion (2002)
[4] Parsons et al (2000)

next section we give an overview of related software toolkits. In Section 3 we discuss the role of simulation modelling in multi-agent systems research. In Section 3 we review work on simulation-modelling of economic scenarios, that is agent-based computational economics. In Sections 4 and 5 we discuss different models of agent interaction and rationality and how they are implemented in JABM. In Section 6.3 we take a step back and give a high-level overview of the design of the JABM by walking through an example model. Finally we conclude in Section 7.

2 Related work

Software for implementing agent-based models has been evolving across several decades and across several disciplines, and many toolkits take an object-oriented approach to modeling. Indeed one of the earliest frameworks, the Swarm system (Minar et al, 1996), was developed in an early object-oriented programming language (Objective-C). One of the disadvantages, however, of Swarm is that Objective-C is no longer widely supported outside of Apple's proprietary operating systems. As a result there have been several frameworks inspired by Swarm that attempt to provide similar object-oriented libraries for implementing agent-based simulations using more modern and widely supported languages. In particular, many have chosen the Java language which provides a unique combination of features which are particular attractive to simulation modellers, viz.: type safety, garbage collection, libraries for numerical methods and high throughput performance (Moreira et al, 2000).

One approach to implementing agent-based models using the Java language is to provide a base toolkit which is extensible through custom scripting environments. This is the approach taken by the popular NetLogo toolkit (Wilensky and Rand, 2011). The scripting approach has many advantages, particularly when using agent-based modeling as an educational tool or for cross-discplinary work where the modelers may not be computer scientists. However, there are also several drawbacks with this approach. Firstly, interpreted scripting languages are not as fast as models implemented directly in Java. Secondly, it is difficult to straightforwardly make use of the many open-source third-party libraries[5] written in Java. Finally, we lose the many advantages of object-oriented modeling when implementing simulations in a non OO scripting language.

One of the most mature Java-based frameworks which takes an object-oriented approach to modeling is the Repast toolkit (North and Macal, 2005; Dubitzky et al, 2011). Repast models, however, are not

[5]e.g. there are many Java libraries implementing neural networks or heuristic optimization algorithms or other numerical methods.

implemented straightforwardly using POJOs. In contrast, toolkits such as MASON (Luke, 2005) adhere more closely to the ideal of implementing agent models using simple objects, and JABM shares much in common with its design philosophy.

Chmieliauskas et al (2012) recently introduced a framework called AgentSpring which is a highly modular and powerful framework for building agent-based models using object-oriented design principles, and also makes extensive use of dependency injection. In AgentSpring, dependency injection is used to promote seperation of concerns between visualisation and persistence functionality, and one of the key innovations of AgentSpring is the use of a graph database (Vicknair et al, 2010) which is used both for persistence, and also allows agents within the simulation to gather data from their environment by running graph-database queries on the attributes of the simulation itself.

Where JABM differs from all of these frameworks is in the unique application of dependency-injection in order to overcome several design problems inherent in executing object-models as if they are Monte-Carlo simulations, with independent realisations of the model being executed with different randomly-drawn values. We will return to this discussion in Section 6.2.

3 Agent-based models in Multi-Agent Systems Research

Multi-agent Systems typically comprise very many autonomous agents with their own local goals or utility functions. Engineering of such systems entails careful reasoning about the collective behaviour of such agents: that is, the macroscopic behaviour of the system as a whole. This is especially difficult since typically the behaviours of individual agents are neither deterministic nor can be entirely pre-specified in advance. This is exactly the problem faced in understanding many problems in socio-economic systems, thus research in multi-agent systems has drawn heavily on theoretical models from economics and game-theory (Shoham and Leyton-brown, 2010), in which we assume that agents are rational expected-utility maximisers whose expectations about the future are model-consistent (the assumption of *rational expectations*).

Despite the many successes of these theoretical models, it is widely known that they sometimes fail to explain and predict phenomena that occur in real-world systems. For example, with the advent of algorithmic trading financial exchanges have become some of the largest and most mission critical multi-agent systems in existence. However, the recent financial crisis highlights the limitations of relying solely on theoretical models to understand these systems without validating them thoroughly against actual empirical behaviour, and it is now acknowledged that widely adopted theoretical models, such as the random walk model of

geometric Brownian motion, are not consistent with the data from real-world financial exchanges (Lo and MacKinlay, 2001).

This had led to a resurgent interest in alternatives to models based on rational expectations models and the efficient markets hypothesis; Lo (2005) proposes the "adaptive markets hypothesis" as an alternative paradigm. The adaptive markets hypothesis posits that incremental learning processes may be able to explain phenomena that cannot be explained if we assume that agents instantaneously adopt a rational solution, and is inspired by models such as the El Farol Bar Problem (Arthur, 1994) in which it is not clear that a rational expectations solution is coherent.

Agent-based models address these issues by modelling the system in a bottom-up fashion; in a typical agent-based model we simulate the behaviour of the agents in the market, and equip them with simple adaptive behaviours. Within the context of economic and financial models, agent-based modelling can be used to simulate markets with a sufficient level of detail to capture realistic trading behaviour and the nuances of the detailed microstructural operation of the market: for example, the operation of the underlying auction mechanism used to match orders in the exchange (Lebaron, 2006).

The move to electronic trading in today's markets has provided researchers with a vast quantity of data which can be used to study the behaviour of real-world systems comprised of heterogenous autonomous agents interacting with each other, and thus a recent area of research within the multi-agent systems community (Rayner et al, 2011, 2012; Palit and Phelps, 2012; Cassell and Wellman, 2012) attempts to take to a reverse-engineering approach in which we build agent-based models of markets that are able to replicate the statistical properties that are universally observed in real-world data sets across different markets and periods of time — the so called "stylized facts" of the system under consideration (Cont, 2001).

A key issue for research in this area is that the way that agents interact and learn can be critical in explaining certain phenomena. For example, LeBaron and Yamomoto (2007) introduce a model of financial markets which demonstrates that certain long-memory characteristics of financial time series data can only be replicated when agents imitate each others' strategies. When their model is analysed under a treatment in which learning does not occur, the corresponding long-memory properties disappear. Performing such analyses requires that we are able to easily reconfigure the way in which agents interact and learn. In the following section we catalog some commonly-used interaction models, and in Section 5 we review models of learning. We return to these topics in Section 6.4, and describe how they can be modelled using OO design, and configured using dependency-injection.

4 Models of agent interaction

A typical agent-based simulation model consists of a population of agents interacting with each other in discrete time. At any given discrete time period, some subset of the population is chosen to interact, during which time each agent can choose a course of action conditional on its observations of the environment and/or other agents depending on the constraints imposed by the model. Each time period is alternatively called a "tick", or a "round", and we will use the former convention in this paper. Ticks represent units of time in the sense that any interactions resulting in state changes to the model that occur within the same tick are considered to have occurred simultaneously.

This basic framework allows for many variants. For example, we might pick randomly chosen pairs of agents from the population to interact with each other during each round in a model akin to evolutionary game theory (Weibull, 1997). Alternatively, we might pick a single randomly-chosen agent on every tick, or allow for a random arrival model in which agents arrive at the simulation with a certain probability as described by a Bernoulli process (that is, a discrete-time approximation of a Poisson process), allow every agent to interact with every other agent simultaneously on each tick, or impose some additional structure on the population so that only certain agents are able to interact with each other.

In this section we attempt to sysmatically catalog different commonly-used conceptual frameworks for representing how and when agents interact with other. We will then turn to models of agent learning and adaptation in Section 5.

4.1 Random pairwise interactions

The simplest framework for modelling interaction between agents assumes that agents do not actively select partners with which to interact, but rather interactions occur due to chance encounters with other agents. This framework is extremely simple — in the simplest case, agents have an equal probability of interacting with any other agent — and its simplicity lends itself to analytic tractability. Hence this is the approach adopted by many mathematical models, the archetypal example being evolutionary game-theory which is used in the field of evolutionary biology to explain coevolutionary adaptations, as pioneered by Maynard-Smith (1973) with the hawk-dove game.

In an evolutionary game-theoretic model, pairs of agents are chosen at random from an idealised infinite population. By assuming an infinite population and no mutation we can specify the dynamics of the system using a simple ordinary differential equation. The standard approach is to use the replicator dynamics

equation (Weibull, 1997) to model how the frequency of each strategy in the larger population changes over time in response to the within-group payoffs:

$$\dot{m}_i = [u(e_i, \boldsymbol{m}) - u(\boldsymbol{m}, \vec{m})]\, m_i \tag{1}$$

where $\boldsymbol{m}$ is a mixed-strategy vector, $u(\boldsymbol{m}, \boldsymbol{m})$ is the mean payoff when all players play $\boldsymbol{m}$, and $u(e_i, \boldsymbol{m})$ is the average payoff to pure strategy i when all players play $\boldsymbol{m}$, and $\dot{m}_i$ is the first derivative of m_i with respect to time. Strategies that gain above-average payoff become more likely to be played, and this equation models a simple co-evolutionary process of adaptation.

In most models which use the replicator dynamics framework, the payoff function $u()$ is expressed as a simple linear equation, typically of the form $u(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x}\boldsymbol{P}\boldsymbol{y}$ where $\boldsymbol{P}$ is the payoff matrix of the underlying game. However, in more complicated models it may not be possible to derive closed-form expressions for this function, and instead we may have to estimate expected payoffs by simulating the interactions between agents and using the average observed payoff as an estimator of the expected value. This methodology is known as *empirical* game-theory, which we will discuss in further detail in Section 5.3.

4.2 Network models

Although the random pairwise framework discussed in the previous section has the virtue of simplicity, it is well known that many agent interactions in reality are highly structured; particular pairs of agents are more likely to interact than others, and in the extreme case this may be a hard constraint such that only certain pairs of agents are able to interact at all.

Mathematically we can express this as a graph structure which specifies the connectivity between agents. Formally a graph is specified by a pair $(\mathbf{V}, \mathbf{E})$ where $\mathbf{V}$ is a set of vertices and $\mathbf{E}$ is a set of edges which connect the edges. Graphs may be either directed or undirected. In the case of a directed graph the edges are ordered pairs of the form $e = (x, y)$ where $e \in \mathbf{E}, x \in \mathbf{V}, y \in \mathbf{V}$. The edge $e = (x, y)$ specifies that vertex x is connected to vertex y but not vice versa. In the case of an undirected graph edges are unordered sets of the form $e = \{x, y\}$ denoting that the vertices x and y are connected to each other. In the context of an agent-based model the vertices represent agents, and the presence of edges typically represents that the corresponding agents can interact with each other.

The importance of network structure has been explored in many disciplines, not only within the multi-agent systems community, but also in economics (Cont and Bouchaud, 2000; Jackson, 2007; Alfrano and

Milakovic, 2009) and biology. An full exposition of the field of network science is outside of the scope of this paper, however see Newman (2010) for a comprehensive recent overview.

The statistical properties of the network structure underlying agent interactions can have dramatic consequences for the behaviour of the system as a whole. One of the most studied properties of networks is the so-called small-world property in which the shortest path distance between randomly chosen pairs of vertices in the network is proportional to the logarithm of the total number of vertices in the network. The prevalence of this property in real-world networks is particularly surprising given that empirically studied networks are also highly *clustered*; if i and j are connected to each other and also x and y, then we have a high chance of also observing that j and y are connected.

The importance of network topology is most notably demonstrated in models of cooperative behaviour which study the conditions under which cooperative outcomes can be sustained by communities of agents who attempt to maximise local objectives. These models are particularly important because they have implications across all many discplines. Santos et al (2006) showed that whether or not cooperation prevails depends on the topology of the network and that small-world networks lead to much greater cooperation. Ohtsuki et al (2006) generalised this result showing that natural selection favours cooperation if the benefit of the altruistic act divided by the cost exceeds the average number of neighbours on the network.

An important consideration in agent-based models which use networked interactions is how to generate the underlying interaction graph. There are several commonly-used graph generation algorithms which yield networks with different statistical properties. The most important of these are the model of Watts and Strogatz (1998), which generates networks that are simultaneously highly-clustered and also have the small-world property, and the algorithm of Albert and Barabasi (2002) which generates graphs with power-law degree distributions.

4.2.1 Adaptive networks

One of the key aspects of many agent-based models is that agents are not merely passive components which update their state unconditionally; rather they are active decision makers and can choose how to interact with the environment, including the other agents present therein. Within the field of evolutionary biology this is called *partner selection* (Noë and Hammerstein, 1995). Once we allow partner selection, the network interactions discussed in the previous section are no longer exogenous static structures, but rather agents can manipulate the network structure in order to maximise their own utility or fitness, leading to potential co-evolution between network structure and agents' strategies (Zimmermann et al, 2000). This has been

explored in several recent models of cooperative behaviour (Do et al, 2010; Phelps, 2012), which show feedback between network structure and agents' strategies plays an important role in determining outcomes.

4.3 Temporal aspects

The previous section discussed different modes of interaction between agents but not *when* they interact. In this section we give an overview of several commonly used frameworks for modelling the progression of time in agent-based modelling.

Agent-based models typically execute a program which represents how the state of some real-world system evolves over a duration of time. However, typically our model will execute more quickly than the corresponding real-world system. We thus need to distinguish between two separate notions of time: simulation-time verses real-time, also known as "wall-clock" time. The former represents the time value available to agents within the simulation, and thus software implementing an agent-based model needs to be able to track the progression of simulation-time and make the simulation clock available to agents whose decisions may depend on time.

Simulation models running on digital computers fall naturally within the discrete event framework (Banks and Carson, 1984) for representing time. In discrete-event simulations we have a countable set of discrete events, each of which has an associated time value which represents the time at which the event is scheduled to occur. Although we have a countable and finite set of events, the time values associated with these events can be real valued numbers; that is, it is important to note that "discrete-event" simulation refers to the set of events rather than time itself, and we can represent continuous time values within such a model.

4.3.1 Arrival process

In the simplest agent-based models an event occurs on every tick of the simulation. However, this does not capture the "bursty" behaviour of real systems in which we typically see many events clustered together in time followed by longer periods of quiescence, along with stochastic variation in the time elapsed between events — the *inter-arrival times*. A more realistic model is the Poisson process (Grimmett and Stirzaker, 2001) in which the inter-arrival times between events are drawn *i.i.d.* from an exponential distribution whose probability densitity function is given by $f(x) = \lambda e^{-\lambda x}$, where λ is the average *arrival-rate* of events per unit time. This can be easily implemented in a discrete-event simulation framework by advancing the simulation clock by the randomly drawn inter-arrival time each time an event is scheduled.

The corresponding discrete process is a Bernoulli process in which an event is scheduled to happen on

any particular simulation tick with a specified probability p. In the limit as the total number of ticks tends to infinity, the Bernoulli process is equivalent to a Poisson process. However, the Bernoulli process is often simpler to implement in software and thus forms the basis of many popular agent-based models in which heterogenous spacing between events is important, e.g. see Iori and Chiarella (2002). We can think of a Bernoulli process as a generalisation of the simplest models in which events occur at every tick with probability $p = 1$; in a Bernoulli model we allow $p < 1$ resulting in stochastic, but discrete, inter-arrival times.

4.3.2 Simultaneous interactions

Continuous-time models based on an arrival process such as those described in the previous section are able to capture statistical features of arrival times that are important for many applications but this realism comes with a price. We often want to view agent interactions as a game of imperfect information in which agents are not necessarily able to observe the actions chosen by the other agents in the model prior to choosing their own action. This can be difficult to capture in an arrival-process model since, by necessity, every event corresponding to an action taken by an agent occurs in a linear sequence, and thus we would have to impose additional constraints on the model in order to prevent agents arriving later than others from observing actions chosen earlier (e.g. because of response-time latencies); in the terminology of game-theory the information-set for each agent remains ambiguous.

What we require in such a scenario is the ability to specify that agents choose their actions simultaneously in a manner akin to a repeated normal-form game such as the iterated prisonor's dilemma (Axelrod, 1997). We can implement this within the framework described at the beginning of this section by specifying that an entire *set* of agents interact with other during a given simulation tick. For example, this set might consist of the entire population of agents — e.g. see Phelps (2012) — or the neighbouring vertices of a particular agent on a network. We can either specify that these simultaneous interactions occur homogenously in time by scheduling them to occur on every tick, or alternatively we can combine this approach with an arrival process model by scheduling the interactions stochastically.

5 Evolution and learning in agent-based modelling

In a typical agent-based model different agents of the same type may be configured with heterogenous behaviours, and a single agent can switch between several different behaviours during the course of a simulation.

Since the outcome of following a specific behaviour can depend on the actions of other agents we use the term strategy from game-theory to refer to behaviours, however we do not dictate that agents use game-theoretic algorithms to choose their strategy since we often want to explore the implications of models in which behaviours are evolved or learnt heuristically. In this section we catalog some of the commonly-used models of learning and adaptation that are used in agent-based models.

5.1 Individual learning

In the simplest type of adaptive agent-based model, agents use a simple reinforcement learning algorithm, such as Q-learning (Watkins and Dayan, 1992), in order to select between their strategies. The central idea is that each agent attempts to estimate the expected payoff through an inductive sampling process in which the agent tries out different strategies and uses the payoff values thus obtained to estimate the expected payoff of each, and hence determine the strategy which will give the best long-term reward – the so-called greedy strategy. Such models have been widely adopted in modelling the behaviour that is empirically observed in strategic environments (Erev and Roth, 1998).

One of the problems with this simple approach that is particularly acute within the context of multi-agent systems and agent-based modelling, is that learning is only guaranteed to converge to the optimal policy provided that the environment is stationary; that is, provided that the expected payoffs do not change over time, and that they do not change in response to the behaviour of the agent. This is patently untrue in the case of a multi-agent system, since the environment consists of other agents who are adapting their choice of strategy in response to each other.

This has led to the development of algorithms designed specifically for multi-agent learning. In a multi-agent context, the notion of an optimal policy is more complicated because the appropriate choice of strategy depends on the behaviour of other agents, who may also be learning. If we consider agents who do not learn, then the appropriate solution concept for such multi-agent interactions is Nash equilibrium (Nash, 1950), in which every agent adopts a best-response strategy to the strategies chosen by other agents who in turn adopt their own best-response strategies. However, if our agent is interacting with agents who are *learning* then the Nash strategy may not be appropriate; for example, in a non-zero-sum game if our opponents are adopting non-equilibrium strategies then we may be able to exploit these agents by adopting a non-equilibrium best-response to their off-equilibrium play.

This has led to refinement of the desiderata for reinforcement algorithms in multi-agent contexts (Shoham and Leyton-brown, 2010, ch. 7). A learning algorithm is *safe* if it is able to guarantee the security value of

the game (the minimum payoff that can be obtained regardless of the strategies chosen by other agents). A learning algorithm is *rational* if it converges to a best-response when its opponents(s) adopt static strategies. Finally the property of *no-regret* specifies that the learning-algorithm is able to gain a superior expected payoff compared with what could have obtained by following any one of its pure strategies.

The development of multi-agent learning algorithms that satisfy one or more of these criteria is an active field of research within the agents community (Bowling, 2005; Busoniu et al, 2008; Kaisers and Tuyls, 2010). However, it is somewhat surprising that these algorithms have not been more widely adopted in the context of agent-based *modelling*, given the importance that the latter community places on adaptive agents. This may be because many of these learning algorithms assume that payoffs for every strategy profile are already known, which is not true in the general case. Instead we may need to estimate or learn the payoffs themselves, as discussed in the following sections.

5.2 Social learning and co-evolution

Heuristic approaches are often used when faced with tractability issues such as these. In particular, heuristic optimisation algorithms, such as genetic algorithms, are often used to model adaptation in biological settings where the fitness of a particular allele depends on the frequency with which other alleles are present in the population — that is, in co-evolutionary settings (Bullock, 1997).

Such models can be implemented using a *Co*-evolutionary algorithm (Hillis, 1992; Miller, 1996). In a co-evolutionary optimisation, the fitness of individuals in the population is evaluated relative to one another in joint interactions (similarly to payoffs in a strategic game), and it is suggested that in certain circumstances the converged population is an approximate Nash solution to the underlying game; that is, the stable states, or equilibria, of the co-evolutionary process are related to the evolutionary stable strategies (ESS) of the corresponding game.

Co-evolutionary algorithms can also be interpreted as models of social learning (Vriend, 2000) in which agents have a probability of switching to another agent's strategy if it is observed to a higher payoff than their current strategy. In such models the operators of the evolutionary algorithm can be interpreted as representing a form of imitation learning; behaviour is learnt through observation resulting in copying (reproduction) with error (mutation). The effect of social learning has been explored in several recent agent-based models of trading behaviour in financial markets (LeBaron and Yamomoto, 2007; Rayner et al, 2012).

However, there are many caveats to interpreting the equilibrium states of standard co-evolutionary algorithms as approximations of game-theoretic equilibria, as discussed in detail by Ficici and Pollack (1998,

2000); Ficici et al (2005). This can sometimes be problematic because, firstly, the equilibria of any given game are invariant under different learning dynamics, and secondly if equilibrium strategy profiles are adopted by learning agents they remain stationary under a wide range of strategy dynamics. Therefore the lack of game-theoretic underpinnings to co-evolutionary models can sometimes call into question the robustness of these models, since results can be highly sensitive to small changes in the learning model and the corresponding dynamics of strategy adjustment. On the other hand, deriving closed-form solutions for the equilibria of any non-trivial agent-based model is intractable in the general case. This has led to the development of methodologies for estimating the equilibria of multi-agent systems using hybrid techniques which combine agent-based simulation, numerical methods and game-theory, which we discuss in the next section.

5.3 Empirical Game Theory

In order to address the issues discussed in the previous section, many researchers adopt a methodology called empirical game-theory (Walsh et al, 2002; Wellman, 2006; Phelps et al, 2010), which uses a combination of agent-based simulation and rigorous game-theoretic analysis. The empirical game-theory method uses a heuristic payoff matrix, which gives the estimated expected payoff to each player adopting a particular pure strategy as a function of the strategies adopted by other players. The methodology is called *empirical* game-theory because we use empirical methods to derive the payoffs; the payoff estimates are obtained by sampling very many realisations of a corresponding agent-based model in which agents are configured with strategy profiles corresponding to the payoff matrix, and we use standard Monte-Carlo methods to derive estimates of the expected payoffs from the sample mean and reduce the variance thereof in order to obtain more accurate comparisons between different payoff values.

The payoff matrix is said to be *heuristic* because several simplifying assumptions are made in the interests of tractability. We can make one important simplification by assuming that the game is symmetric, and therefore that the payoff to a given strategy depends only on the *number* of agents within the group adopting each strategy. Thus for a game with j strategies, we represent entries in the payoff matrix as vectors of the form $\boldsymbol{p} = (p_1, \ldots, p_j)$ where p_i specifies the number of agents who are playing the i^{th} strategy. Each entry $\boldsymbol{p} \in P$ is mapped onto an outcome vector $\boldsymbol{q} \in Q$ of the form $\boldsymbol{q} = (q_1, \ldots, q_j)$ where q_i specifies the expected payoff to the i^{th} strategy.

For a game with n agents, the number of entries in the payoff matrix is given by $s = \frac{(n+j-1)!}{n!(j-1)!}$. Although it is still intractable to estimate the heuristic payoff in the general case, for small numbers of agents and strategies — for example, for $n = 10$ agents and $j = 5$ strategies, we have a payoff matrix with $s = 1001$

entries — we can obtain heuristic payoff matrices that are sufficiently precise to give insights into many strategic interactions that occur in multi-agent systems (Jordan et al, 2007; Phelps et al, 2009; Cassell and Wellman, 2012).

6 Object-oriented agent-based modelling

The underlying design philosophy of JABM is that object-oriented programming languages provide a natural way of expressing all of these different conceptual frameworks for modelling agents, their interactions and how they learn or evolve. For example, agents can be represented as instances of objects[6], heterogeneous populations can be represented by different classes of object, and the properties of agents correspond directly with class attributes. Temporal aspects of the model can be represented within a discrete-event simulation framework (Banks and Carson, 1984) and once again events can be modelled as objects with different types of event being represented by different classes. Describing agent-based simulation models in terms of objects is highly intuitive and leads to natural implementation in object-oriented programming languages. By representing entities such as agents, events and even the simulation itself as objects we automatically grant these entities status as "first-class citizens" (Burstall, 2000) within whatever OO programming language we use.

JABM uses objects to model the different approaches to agent interaction and learning described in Sections 4 and 5. The modular design allows the modeller to reconfigure experiments to use these different approaches to interaction and learning without making any changes to the code representing the agents themselves.

In the following section we describe the basic object-oriented architecture for implementing discrete-event simulation. In Section 6.2 we proceed to give an overview of the dependency injection design pattern, and show how it can be used for Monte-Carlo simulation. In Section 6.3 we give an overview of the design of the JABM toolkit. Finally, in Section 6.4 we describe how learning and evolution are modelled using objects.

6.1 Discrete-event simulation using object-oriented programming

Discrete-event simulation models are naturally implemented using techniques from object-oriented programming (Garrido, 2001). Most object-oriented frameworks for agent-based modeling represent individual agents

[6]Note that the focus of our work is the use of agent-based simulations to *model* complex systems. This entails fundamentally different design requirements from software designed to *implement* agent systems or multi-agent systems (Chevaleyre et al, 2006); agent-oriented software engineering is outside of the scope of this paper and in the remainder of the discussion we shall focus on agent-based modeling.

as object instances whose attributes correspond to the properties of the agents in the model. Heterogeneous populations in which we have different types of agents can then be easily modelled by using different classes of object, and base "skeleton" or "default" code required to implement agents can be provided by inheritance, for example in the form of abstract classes.

Discrete-event processing is naturally implemented using the Observer or Model/View/Controller design patterns (Gamma et al, 1995) in which events are themselves modelled as objects, and different types of event are represented by different classes of object. Each event has an attribute corresponding to a time-stamp (that is, the tick number at which the event was generated), as well as any other state information pertinent to the particular class of event. Various entities in the simulation can then listen or "subscribe" to the particular events that they are interested in and respond accordingly. For example, in a simulation of an electronic marketplace such as an open-outcry ascending auction, an agent might subscribe to the class of event that represents being outbid, and respond to this event by raising its own bid where appropriate. These design patterns lead to a highly modular and flexible design which has three principal benefits, as detailed below.

Firstly, reporting, analysis and visualisation functionality can be cleanly separated from code representing the agent model itself. This allows re-use of these components across different models (for example, a View component showing a histogram of the proportion of different types of agent in the simulation at any given moment in time can be deployed in *any* agent-based model), and also allows the modeller to deploy various levels of visualisation or reporting depending on the trade-off required between CPU time and the amount of data required for analysis (for example, visualisation can easily be turned off in order to speed up the simulation). Reporting functionality also becomes straightforward to implement: a report is itself an object which listens to events, and can then calculate statistics to be logged to a file or database – for example, the average price of a bid during each auction can be calculated by listening for bid events and then calculating the average price using the `price` attribute of the event.

Secondly, these design patterns encourage the simulation modeller to use object-oriented programming in the manner in which it was originally conceived, viz. to use classes to encapsulate the properties and behaviour of entities that exist in the problem domain. For example, in a non-simulation context we might create a class called `Book` to represent an actual real book in a electronic library catalogue system, which attributes such as `title`, `author` etc., and methods such as `checkOut`, `checkIn` etc. The object-oriented approach to designing software systems is to treat the software application as a *model* of real-world entities, and hence this style of programming is naturally suited to agent-based modeling. For example, in an

agent-based model of an auction we might create a class called `BiddingAgent` with attributes such as `name`, `utilityFunction`, `valuation` and methods such as `placeBid`.

Finally, the loose coupling between originators of events and receivers of events allows us to "rewire" the basic underlying components of our simulation in different ways depending on the specific agent model we are trying to implement. For example, returning to our auction model, the `Auctioneer` agent might listen for occurrences of `BidPlacedEvent` in order to update the auction state. However, we might we might then reconfigure the model by allowing `BiddingAgent` objects to also listen to `BidPlacedEvent` in order to implement a bidding strategy which is conditional on the behaviour of other agents in the market.

Often we would like to be able to rewire the model in this way very easily in order to perform experiments under different treatment conditions: for example a *sealed bid* auction in which only the auctioneer has access to bids verses an *open outcry* auction (Krishna, 2002) in which the bids of all agents are publicly available. Ideally we would like to perform this configuration without having to change any of the code representing the entities in our model, since we are simply changing the structure of the *dependencies* between these entities as opposed to the entities themselves. This can be achieved using the Dependency Injection design pattern (Fowler, 2004; Prasanna, 2009), in which the dependencies between objects are expressed declaratively and are then "injected" into our model by a separate third-party container: a form of Inversion of Control (IoC).

We discuss this design pattern in detail in the next section, and show how it is particularly useful for simulation models.

6.2 Dependency Injection

This section describes how several design problems that arise in implementing agent-based models in the form of simulation software are naturally solved by a software engineering design pattern (Gamma et al, 1995) called "dependency injection" (Fowler, 2004; Prasanna, 2009) which promotes "separation of concerns" between the configuration of an object model and the implementation thereof.

As discussed in the previous section, the underlying design philosophy of JABM is that object-oriented programming languages provide many of the primitives necessary for agent-based modeling.

However, some aspects of agent-based models are difficult to encapsulate directly in an OO framework. In particular, any non-trivial agent-based simulation model will be *stochastic* in nature: many of the attributes of the objects in our model will be random variables which take on different values each time the simulation is executed. Alternatively, we might configure these same attributes as independent variables to be controlled in a "parameter sweep"; again this results in certain object attributes taking on different values each time

the model is executed.

The fact that agent-based models are often executed as Monte-Carlo simulations poses difficulties for our principle that models be represented as object-oriented programs because variables in OO programming languages are *not* random; random and/or independent variables are not first class citizens in most modern programming languages. One work-around for this issue has been to simply implement code, for example in each object's constructor, which initialises any free parameters from various probability distributions using a Pseudo-Random Number Generator (PRNG). We can then create new instances of the objects representing the entities in our model on each independent run of the simulation, ensuring that new random variates are assigned on each run whilst maintaining state that needs to persist across different runs (for example, objects representing summary statistics) using singleton objects (Gamma et al, 1995, p. 127).

However, such an approach is not declarative and results in a situation where if we want to run different experiments in which we use different probability distributions, or switch from using randomly-sampled variables to controlling a variable explicitly, we have to modify the objects representing the model itself.

In contrast, the dependency-injection design pattern allows us to separate these concerns. Configuration information specifying how the objects in our model are to be initialised and the run-time dependencies between them are expressed declaratively, and then "injected" into the application at runtime. This allows us to write code representing, for example, the agents in our model without having to make any assumptions regarding how the variables representing the agent's properties are to be initialised. Thus we can run execute the same model under a wide variety of experimental treatments without modifying the model itself.

In the remainder of this section we describe how dependency injection has been applied in the design of the JABM toolkit.

6.2.1 Monte-Carlo Simulation

In the previous section we discussed how agent-based models in the form of discrete-event simulations can be naturally viewed as object models, and straightforwardly implemented in an object-oriented programming language. However, there are some aspects of the discrete-event simulation methodology that do not have corresponding constructs in typical object-oriented programming languages. In particular, when performing an experiment using a simulation model we typically execute the simulation very many times in order to account for the fact that any non-trivial model is non-deterministic, and also to examine the sensitivity of the model to random variations in free parameter settings, or to explore the response of certain macroscopic quantities to changes in the underlying parameters; for example, by performing a controlled experiment in

Listing 1: Initialisation of random variables without dependency injection

```
public class BiddingAgent {

        /**
         * The agent's private valuation for the item
         * it is currently bidding on.
         */
        double valuation;

        double stdevValuation = 1;

        double meanValuation = 100;

        public BiddingAgent(Random prng) {
                this.valuation = prng.nextGaussian()
                                * stdevValuation + meanValuation;
        }

}
```

which systematically increase an independent variable to see how it influences other dependent variables.

In order to achieve this, we might for example run the same model many times, and on each run we initialise the values of certain parameters of the model (represented as object attributes) by drawing a value from a pre-specified probability distribution. That is, on each run we initialise the random variables of our model to particular randomly-drawn values ("random variates").

One way of achieving this is to simply write code in the objects representing entities in our model; for example we might initialise the `valuation` attribute of our `BiddingAgent` object to a random value in its constructor as illustrated in Listing 1. However, this approach is not ideal if we want to explore the implications of changes in the probability distribution of this variable to the outcome of the auction. For example, we might want to set up one experimental treatment in which agents' valuations are independently drawn from a common distribution verses another in which the valuations of different agents are drawn from different distributions, or to explore the sensitivity of outcomes to the type of distribution used (for example, normal verses uniform). In order to do this, we would need to modify the Java code in Listing 1 for each treatment. From a software-engineering perspective, this is a potential source of defects in the model since we risk introducing bugs into the code representing an agent each time we make a modification. Ideally we would like some way to configure each of this experimental treatments without having to change any of the code representing the model itself.

The solution is to configure this object object by *injecting* its parameters, thus decoupling the configura-

Listing 2: A plain-old Java Object (POJO) representing an agent which will be initialised using dependency injection

```
public class BiddingAgent {

        /**
         * The agent's private valuation for the item
         * it is currently bidding on.
         */
        double valuation;

        public double getValuation() {
                return valuation;
        }

        public void setValuation(double valuation) {
                this.valuation = valuation;
        }

}
```

tion of the model form its actual implementation. The first step is to refactor the object as a Plain Old Java Object (POJO), which is simply an object with a zero-argument constructor which receives its configuration passively via setter and getter methods (Parsons et al, 2000; Richardson, 2006). Listing 2 shows the POJO corresponding to our agent.

The second step is to delegate responsibility for configuring this POJO to a separate container. In our case, we will use the industry-standard Spring framework (Johnson et al, 2011) to configure the objects making up the agent-based model. A full overview of Spring is outside of the scope of this paper, but the central concept is very simple. All of the objects in our model are constructed using a separate object factory[7] provided by the Spring framework. This factory, called the "bean factory", uses an Extensible Markup Language (XML) file to configure the objects required by our simulation model. In Spring, each object instance in the simulation is called a "bean" and is manufactured and configured by Spring's bean factory. The XML file, called the "Spring beans configuration file", has a tag for each bean which allows us to configure its properties, including any dependencies with other beans.

As a simple example, consider the POJO for our `BiddingAgent` in Listing 2. Listing 3 shows one way we might configure this object using dependency injection. Here we specify that we want to construct a single object instance (which we refer to as "myAgent") of the class `BiddingAgent`, and we want to set its `valuation` attribute to the value `10.0`. When our model is initialised by the Spring framework, the bean

[7](Gamma et al, 1995, p. 87)

Listing 3: Initialising a POJO using dependency injection and a Spring beans configuration file

```
<bean id="myAgent" class="BiddingAgent">
        <property name="valuation" value="10.0"/>
</bean>
```

factory will create a new instance of `BiddingAgent` using the zero-argument constructor, and then proceed to configure the `valuation` attribute by invoking the corresponding setter method.

Although this example is very simple, it illustrates a very important principle, viz. separation of concerns between the configuration of the model and the model itself. The model, in our case the `BiddingAgent` class, does not "know" or "care" how it is being configured, and this loose coupling allows us to configure it in a myriad of different ways without having to change any of the code representing our model. Listing 4 illustrates a more complex example in which we configure two separate agents to use randomly-drawn valuations independently drawn from the same underlying uniform probability distribution. Again, this situation is modelled straightforwardly using object-oriented principles: different types of probability distribution can be represented as different classes[8] which can themselves be configured using dependency injection.

We can then "wire up" the dependencies between the different components of our simulation, since each bean has unique name which we can refer to from within the definition of another bean. In Listing 4 we have two agents represented by the beans `agent1` and `agent2`, and in contrast to the previous example we configure the `valuation` attribute so that it takes on a *random* value. This is represented by creating a new bean definition, `valuationRandomVariate` which encapsulates the concept of a random variate using an object factory: new values are constructed from the probability distribution specified by the `distribution` attribute, which in turn refers to another bean `commonValuationDistribution`. This bean specifies how we construct the class representing the probability distribution, in this case a Normal distribution $N(100, 1)$ using the Mersenne Twister algorithm (Matsumoto and Nishimura, 1998) as the underlying PRNG.

Notice that we have performed all of this configuration without having to change any of the code representing our agent (Listing 2). Instead, we have represented the different components of our model as different objects, and we have specified the dependencies between these objects declaratively by using the `ref` keyword in our Spring configuration file. These dependencies are then *injected* into our model by the bean factory without having to hard-code them into the code representing our simulation model. This allows us to easily re-configure the components of our model for different experiments. For example, we can easily

[8] In this case the classes representing different probability distributions are provided by the CERN colt library (Binko et al, 2004)

Listing 4: Initialising random variables using dependency injection and a Spring beans XML configuration file

```
<bean id="agent1" class="BiddingAgent" scope="prototype">
        <property name="valuation" ref="valuationRandomVariate" />
</bean>

<bean id="agent2" class="BiddingAgent" scope="prototype">
        <property name="valuation" ref="valuationRandomVariate" />
</bean>

<!-- This bean takes different values each time it is referenced -->
<bean id="valuationRandomVariate"
      class="net.sourceforge.jabm.spring.RandomDoubleFactoryBean"
      scope="prototype">
  <property name="distribution" ref="commonValuationDistribution"/>
</bean>

<!-- The common probability distribution used to draw agents valuations -->
<bean id="commonValuationDistribution" scope="simulation"
        class="cern.jet.random.Normal">
   <constructor-arg value="100.0" />
   <constructor-arg value="1.0" />
   <constructor-arg ref="prng" />
</bean>

<!-- The Pseudo-Random Number Generator (PRNG) used to
        generate all random values in the simulation -->
<bean id="prng" scope="singleton"
   class="cern.jet.random.engine.MersenneTwister64">
</bean>
```

run the model in an experimental treatment in which we treat agents' valuations as an independent variable to be explicitly controlled, as per Listing 3, and then use a different configuration file to run the same model where we treat valuations as a random variable in a Monte-Carlo simulation as per Listing 4. It is also trivial to modify the configuration so that different agents have different valuation distributions (thus enabling the experimenter to explore the implications of deviating from the symmetric-independent private-valuations assumptions of conventional auction theory).

6.2.2 Dependency-injection and multiple runs

Dependency-injection is particular attractive for experiments involving Monte-Carlo simulation because in a typical simulation experiment we execute the model very many times, which presents some subtle design issues. On the one hand, each run of the simulation requires that we initialise the agents in our model afresh, drawing new values of random variables independently from previous runs. On the other hand, however, it is important that some components of our simulation retain state across different runs: for example, we may want to collect summary statistics on certain dependent variables in our model in order to estimate their

expected value, and the PRNG itself must have persistent state across runs in order to prevent spurious correlation in random variates.

Dependency-injection and IoC provide a natural solution to these issues. All of the components of our simulation are managed by the bean factory, and we can request new instances of these objects from the factory on each run of the model. Beans can be configured as "prototype" beans, meaning that each time they are referenced the container will construct a fresh instance from the factory. Alternatively, they can be configured as "singleton" beans meaning that only a single instance of the object is created and the same instance is returned, as per the singleton design pattern (Gamma et al, 1995, p. 127). For example, in Listing 4, the `prng` bean is declared as a singleton using the `scope="singleton"` attribute of the bean tag. On the other hand, the beans representing our agents and the bean `valuationRandomVariate` are declared using `scope="prototype"`, meaning that each time we request an agent from the bean factory, we will obtain a newly-constructed `BiddingAgent` instance with a new random value for its `valuation` attribute. Meanwhile, the PRNG used to draw this random value will persist across different runs in virtue of its singleton scope.

The Spring framework also allows us to define custom scopes. JABM defines a new scope "simulation" which specifies that the bean is a singleton for the duration of an individual simulation run, but should be reinstantiated for the next run. Thus the `commonValuationDistribution` bean in Listing 4 will be constructed afresh for each run of the simulation.

Much of the essence of this design could have been achieved with various singleton and factory classes hard-coded in Java. However, by specifying the dependencies declaratively we can achieve a level of loose-coupling between our components that allows us to easily reconfigure our simulation, and avoid some of the intrinsic design problems inherent in the use of singletons (Rainsberger, 2001). Because configuration is injected into our model, we can set up different experiments, making use of different random distributions for whatever attributes of our model we like, *without having to create or modify any Java code.*

The ability to configure our simulation is further facilitated by additional post-processing hooks made available from the Spring framework. The JABM framework provides the ability to configure the attribute of any bean as a random variable using more user-friendly syntax than that provided in the raw Spring beans configuration file. Listing 5 shows an example JABM configuration file. This configuration is used in conjunction with a Spring beans configuration file but serves to override the values of any corresponding bean attributes (it is applied in a post-processing step when fetching a bean from the factory). In this example, the `valuation` attribute of the bean `agent1` will be initialised from a uniform distribution on the interval

Listing 5: Configuring random variables using a properties file in JABM

```
# Configure the valuation attribute of an agent as a random variable
#  drawn from a Uniform distribution.

agent1.valuation = U(50,100)
```

Listing 6: Configuring random variables and constant parameters in the same properties file

```
# Run the simulation over 1000 independent runs
simulationController.numSimulations = 1000

# Our population consists of 100 agents
population.numAgents = 100

# Configure the valuation attribute of an agent as a random variable
#  drawn from a Uniform distribution.
agent.valuation = U(50,100)
```

$(50, 100)$.

We can also mix constant values and random distributions in the same configuration file, as illustrated in Listing 6. In this example the simulation will be run with a constant number of agents: 100, but we will run the simulation 1000 times drawing agents' valuations from a uniform distribution $\sim U(50, 100)$ on each run.

Alternatively we can treat particular attributes of our model as independent variables to be analysed under a parameter sweep. Listing 7 shows such an example in which we perform $9 \times 2 = 18$ independent experiments by systematically varying both the `learningRate` and `discountRate` attributes of a bean called `qLearner` which represents an implementation of the Q-Learning reinforcement-learning algorithm (Watkins and Dayan, 1992). In this case, the Q-learning parameters will take on all combinations from the set $\{0.1, 0.2, \ldots, 0.9\}$ for each independent experiment, allowing us to analyse the effect of varying these parameters over the specified range.

The important thing to note about these examples is that this configuration is *injected* into our model. Nowhere in our agent-based model do we have to write Java code to read and parse the configuration files and then set the relevant object attributes. Nor do we have to duplicate our object model by re-declaring a mapping between identifiers in the configuration files and attributes in our object model. Rather, we simply

Listing 7: Performing a parameter sweep

```
qLearner.learningRate = 0.1:0.1:0.9
qLearner.discountRate = 0.1:0.1:0.9
```

create POJOs for the entities in our model (Listing 2), and the dependency injection framework takes care of initialising object attributes transparently behind the scenes using Java reflection. The components of our agent based model are fully specified by their natural implementation as objects, and dependency injection tools such as Spring and JABM allow us to wire up these components in different configurations depending on the particular experiment we are performing. Regardless of whether we treat certain object attributes as random variables or as independent variables in a parameter sweep, the actual code representing the agents in our model remains the same.

6.3 Putting it all together: design overview

In this section we give a high-level overview of the design of JABM illustrating how a simple agent-based model can be recursively configured using dependency-injection by walking through some of the key aspects of the configuration of one of the example models which ships with the distribution, viz. the El Farol Bar model (Arthur, 1994).

As discussed, the design philosophy of JABM is to use POJOs to represent the various entities in our agent-based model. These POJOs are configured via dependency-injection by building on the industry standard Spring framework (Johnson et al, 2005) which is packaged as part of the JABM distribution archive[9]. These components are organised into packages as per Table 6.3, and Figure 6.3 shows the relationship between the key classes as a Unified Modeling Language (UML) diagram. JABM allows us to "wire up" these classes in different configurations using dependency injection as described in the previous section. In the remainder of this section we illustrate how we can configure these components to implement the El Farol Bar model.

The top-level component of JABM is the class `SpringSimulationController` which represents a Monte-Carlo experiment in which we execute one or more independent simulation runs on which we collect data. As with any other component in our model it is configured using dependency-injection; Listing 8 shows an example Spring beans configuration file from the JABM distribution which provides an implementation of the El Farol Bar model introduced by Arthur (1994). In this agent-based model, a population of 100 agents decide whether or not to attend a bar which has a capacity of 60. Each agent decides to attend if and only

[9]We also include several Java libraries which are useful in implementing agent-based models, and which are also widely used in other simulation modeling toolkits: the Java Universal Network/Graph Framework (JUNG) library (O'Madadhain et al, 2011), which is used for analysing and visualising social network models; the Apache Commons Math library (Andersen et al, 2011) which provides functionality for calculating summary statistics, several heuristic optimisation algorithms and implementations of many numerical methods; the CERN Colt library[10] which contains an implementation of the Mersenne Twister algorithm (Matsumoto and Nishimura, 1998) and also encapsulates various probability distributions useful in Monte-Carlo simulation; and finally JFreeChart (Gilbert, 2000) which provides graphing functionality. These libraries are distributed with JABM without modification, and they can be used in their distributed form independently of the JABM classes in keeping with our minimialist philosophy.

`net.sourceforge.jabm`	Classes representing the simulation itself
`net.sourceforge.jabm.agent`	Classes representing agents, and populations thereof
`net.sourceforge.jabm.strategy`	Classes representing the behaviour of agents
`net.sourceforge.jabm.event`	Classes representing simulation events
`net.sourceforge.jabm.learning`	Classes representing reinforcement-learning algorithms
`net.sourceforge.jabm.evolution`	Classes for modelling social learning and evolution
`net.sourceforge.jabm.gametheory`	Classes for empirical game-theory experiments
`net.sourceforge.jabm.mixing`	Encapsulation of different agent mixing schemes
`net.sourceforge.jabm.report`	Reporting functionality

Table 1: The main packages provided by JABM

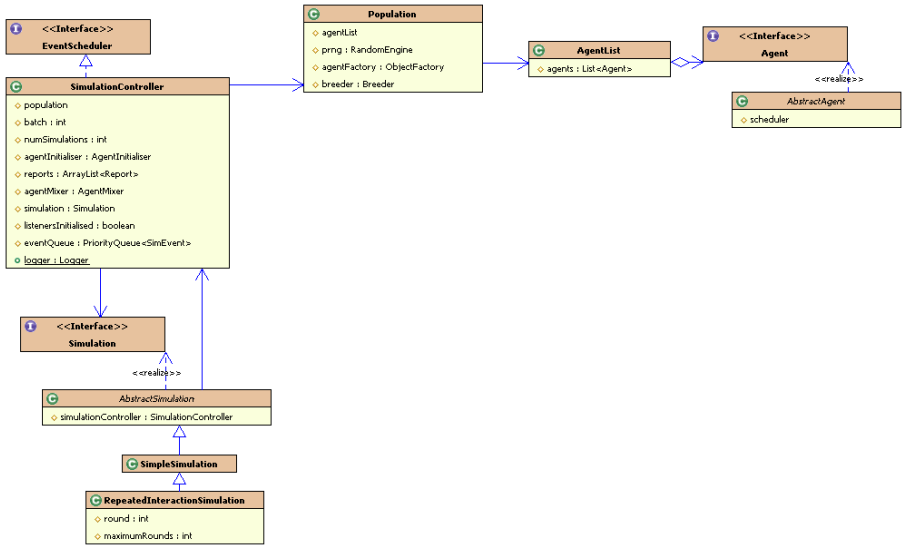

Figure 1: UML diagram showing the key components of JABM

if they forecast they fewer than 60 agents will attend the bar in the next period. In this scenario, it is not clear that a rational expectations equilibrium applies, thus motivating the use of inductive heuristics which predict future attendance based on an analysis of attendance in previous periods. The emergent "adaptive expectations" dynamic equilibrium (Lo, 2005) which arises from agents using inductive strategies which are learnt over time gives rise to complex time series with qualitatively similar features to those found in many empirical economic time series data. Figure 2 illustrates the attendance at the bar as a time series as produced by JABM's `TimeSeriesChart` class.

In the above example we configure three attributes of our experiment. The `numSimulations` property determines the number of independent simulation runs to be executed: in this case we will run the simulation 100 times, drawing random variates independently on each run. This is achieved by repeatedly requesting a new bean from Spring's bean factory which represents the underlying simulation. The `simulationBeanName`

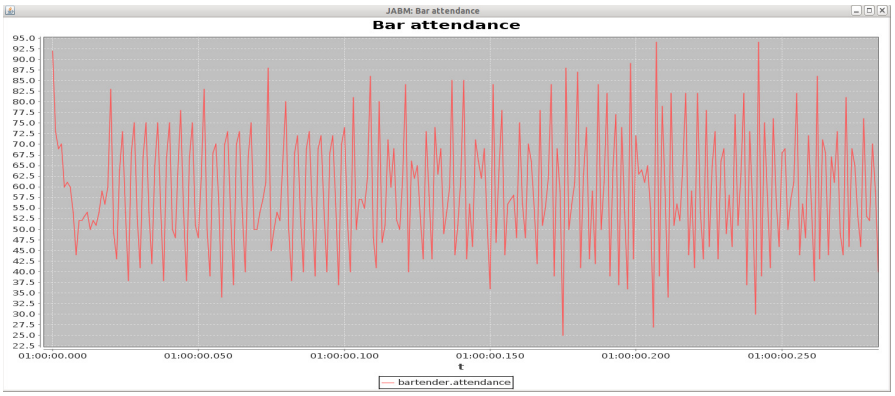

Figure 2: The number of agents attending the El Farol Bar as a time series chart, produced using the `TimeSeriesChart` object in the JABM toolkit. This class allows the modeler to produce interactive charts of statistics produced during simulation runs using the Model/View/Controller design pattern with the view being provided by the JFreeChart library (Gilbert, 2000), and the model provided by any simulation object which implements JABM's `ReportVariables` interface.

Listing 8: Configuring a simulation experiment

```
<bean id="simulationController"
   class="net.sourceforge.jabm.SpringSimulationController">

     <!-- Run the underlying simulation 100 times -->
     <property name="numSimulations" value="100" />

     <!--  This is the name of the bean representing the simulation.
           If we run the simulation more than once the bean will be
            reinstantiated on each simulation run. -->
     <property name="simulationBeanName">
       <idref local="repeatedSimulation" />
     </property>

     <!--  Report objects collect data on the simulation runs.
      Reports persist across simulation runs and are
      singleton in scope allowing them to collect summary
      statistics across different simulations. -->
     <property name="reports">
       <list>
            <!--  The barTender tracks current
                    and historical statistics on attendance -->
               <ref bean="barTender"/>

            <!--  Log attendance to CSV files -->
               <ref bean="attendanceCSVReport"/>

            <!-- More reports can be configured
                  here as required -->
         </list>
     </property>
</bean>
```

Listing 9: Configuring the underlying simulation

```
<!--  The El Farol Bar will be run as a repeated game with 1000 rounds,
          each round representing a week in simulation time.
         Note that this bean is defined as a prototype so that
         when the simulation is run many times, we get a freshly
         constructed simulation object on each run. -->
   <bean id="repeatedSimulation" scope="prototype"
         class="net.sourceforge.jabm.RepeatedInteractionSimulation">
       <property name="maximumRounds" value="1000" />
       <property name="population" ref="population" />
       <property name="agentInitialiser" ref="agentInitialiser" />
       <property name="agentMixer" ref="randomRobinAgentMixer" />
       <property name="simulationController" ref="simulationController" />
   </bean>
```

attribute specifies the name of this bean (which is defined elsewhere in the configuration). When this bean is reinstantiated by the Spring container other beans that it references will also be reinstantiated, resulting in new values of random variates being injected into class attributes as detailed in Section 6.2.

The underlying simulation used in our experiment is represented by the bean `repeatedSimulation` shown in Listing 9. Here we specify the agents in the model, how they interact and how often. The `agentMixer` attribute specifies how agents interact; in this example every agent interacts with the simulation during each tick, but the ordering of agents is randomized to eliminate any artifacts that could arise from polling agents in a particular sequence. Once again, we can easily reconfigure our simulation to use one of the alternative agent mixing models discussed in Section 4 by specifying a different dependency for this attribute.

The population of agents is specified by the `population` bean shown in Listing 10. Here the number of agents in the population is specified by the `size` attribute, and in our example we configure our model to use a population of 100 agents. Each agent is constructed from the specified prototype bean `patronAgent` and here we can specify the class used to represent our agents and then configure their attributes. In our example, we configure the `barCapacity` attribute, which represents the overcrowding threshold, to be the same value 60 for all agents, and we configure the behaviour of our agents by specifying the `strategy` attribute: here we wire-up our agent to another bean `adaptivePredictionStrategy` which represents a learning-classifier system for predicting future bar attendance by selecting amongst different strategies for making forecasts of future attendance based on previous attendance levels (we return to a discussion of learning in JABM in the following section).

Finally, returning to our top-level bean `simulationContoller` in Listing 8 we configure the `reports` property which specifies the objects that will collect data on the simulation runs: in this case the `barTender` bean will track current and historical attendance during an individual simulation run and the `attendanceCSVReport`

Listing 10: Configuring the population of agents and their strategies

```
<!-- The population consists of 100 patrons -->
<bean id="population" class="net.sourceforge.jabm.Population"
    scope="prototype">

     <!-- The factory used to construct agents -->
     <property name="agentFactory">
         <bean
           class="org.springframework.beans.factory.config.ObjectFactoryCreatingFactoryBean"
               >
               <!-- The name the agent prototype bean -->
               <property name="targetBeanName">
                  <idref local="patronAgent" />
               </property>
         </bean>
     </property>

     <!-- The numbner of agents in the population -->
     <property name="size" value="100" />

     <!-- The Pseudo-Random Number Generator (PRNG)
           used to pick an agent at random
           from the population -->
     <property name="prng" ref="prng" />
</bean>

<!--  The prototype used to manufacture patron agents -->
<bean id="patronAgent" scope="prototype"
   class="net.sourceforge.jabm.examples.elfarolbar.PatronAgent">

      <property name="strategy" ref="adaptivePredictionStrategy" />
      <property name="barCapacity" value="60" />
      <property name="scheduler" ref="simulationController" />

</bean>
```

bean is responsible for logging the data from these runs to Comma Separated Values (CSV) files which can then be imported into mathematical analysis tools such as R or MATLAB. Additionally, we can optionally add other report objects to this list, for example a `TimeSeriesChart` to produce an interactive chart of the attendance time series as the simulation progresses, as illustrated in Figure 2. An important feature of JABM is that such functionality can easily be deconfigured if the model needs to be run "headless" mode, for example on a cluster. This can be achieved simply by commenting out the relevant report from the beans configuration file and does not require any change to the code representing the simulation model itself.

6.4 Learning and evolution in JABM

In this section we give a brief overview of how the different models of learning and evolution described in Section 5 are implemented in JABM. As previously discussed, we use the term "strategy" to refer to a particular behaviour of an agent. Within any agent-based model, agents may change their behaviour over time using the various different approaches outlined in Section 5.

Because we are taking an object-oriented approach, it is natural to model the various strategies that an agent can use as different classes implementing a common interface. JABM provides an interface called `Strategy` within the package `net.sourceforge.jabm.strategy`, and any class which implements this interface can be used to define an agent's behaviour in a JABM simulation. For example, within the context of the El Farol Bar problem, we define an abstract class called `AbstractPredictionStrategy`. The various different types of forecasting rules described by Arthur (1994) are then implemented as different sub-classes of `AbstractPredictionStrategy`. For example, we have an `AutoRegressivePredictionStrategy` which makes forecasts using a linear auto-regressive model, and `ConstantPredictionStrategy` which makes a constant forecast in each time period.

Now that we have defined the possible strategies for our agents, we can specify how, and if, agents adapt these strategies in response to a changing environment. JABM provides several additional packages containing classes that can be used to configure adaptive behaviour. The `net.sourceforge.jabm.evolution` package provides functionality for implementing social learning models of the type described in Section 5.2. We can do so by configuring our simulation with an `EvolvingPopulation` instead of a `Population`. We then configure our `EvolvingPopulation` with a `Breeder` which specifies how the population reproduces. One type of `Breeder` is a `FitnessProportionateBreeder`. This implements fitness proportionate selection and can be configured with a `FitnessFunction` to calculate the fitness of an agent's strategy.

Note that this learning functionality is completely decoupled from the code representing the strategies

themselves. This means that we can very easily reconfigure our agent-based model to use a different type of learning. For example, we can turn off the social learning described above simply by reconfiguring our simulation to use a `Population` of agents instead of an `EvolvingPopulation`. Moreover, dependency injection allows us to do so without making any changes to the code representing the model itself; in our example XML configuration file we can simply change the referent of the `population` attribute of the `repeatedSimulation` bean so that it refers to a bean defining an `EvolvingPopulation`.

JABM also provides packages for implementing reinforcement-learning; the `net.sourcerorge.jabm learning` package provides implementations of several commonly used reinforcement learning algorithms which are encapsulated by the `Learner` interface. Agents can then be configured with a strategy `RlStrategy` which in turn can be configured with a specific `Learner`. `RlStrategy` is a meta-strategy; it adheres to the `Strategy` interface and can be configured with a set of underlying "pure" strategies over which it will perform action selection according to the specified learning algorithm. Once again, we can reconfigure our model to use a particular reinforcement learning algorithm without making any changes to our agents or the code representing the underlying strategies.

Finally, the empirical game-theory methodology described in Section 5.3 is implemented by classes in the `net.sourceforge.jabm.gametheory` package. By configuring a `GameTheoreticSimulation` we can automatically set up an experiment to estimate the expected payoffs for every strategy profile in a heuristic payoff matrix. The payoff matrix can then be exported to, e.g. MATLAB, for further analysis. This is the approach that was used in (Phelps et al, 2009); JABM was used to conduct agent-based simulations for each strategy profile and the resulting payoff matrix was used to iteratively solve the replicator dynamics equation (eq. 1) using MATLAB's ODE toolbox in order to produce phase portraits of the evolutionary dynamics of the model.

7 Conclusion

In this paper we have given an overview of agent-based modelling, and cataloged some commonly used frameworks for modelling agents' learning and interaction. We have shown how these frameworks can be modelled using an object-oriented approach, and introduced the Java Agent Based Modeling (JABM) framework which synthesises these different approaches within a common framework.

We have given an overview of dependency-injection and Inversion of Control (IoC), and shown how it can be used to solve several related design problems when implementing agent-based models in the form

of discrete-event simulation software; simulation models are naturally implemented using an object-oriented design in which different types of entity in our model are represented by different classes of object, and individual agents or events as instances thereof. This motivates a simple architecture using, for example, Plain Old Java Objects (POJOs) to represent our model; a design philosophy we have followed in the implementation of the JABM toolkit.

However, the fact that agent-based models are typically executed as Monte-Carlo simulations in which we reinitialise the model with different random variates or control treatments on execution of the model makes it difficult to model simulation entities using a straightforward POJO architecture. In this paper we have shown how dependency-injection frameworks such as Spring solve this problem by separating configuration concerns from the underlying object model. Building on Spring, JABM provides functionality for specifying which attributes of our objects are to be treated as control parameters or random variates, and can then execute the underlying simulation by automatically injecting random variates into the model on each run. This prevents cross-contamination between code representing experiments or treatments and code representing the model itself, and thus allows researchers to concentrate their effort on the latter.

References

Albert R, Barabasi A (2002) Statistical mechanics of complex networks. Reviews of modern physics 74:47–97

Alfrano S, Milakovic M (2009) Network structure and N-dependence in agent-based herding models. Journal of Economic Dynamics and Control 33(1):78–92, DOI 10.1016/j.jedc.2008.05.003

Andersen MM, Barker B, Chou AD, Diggory M, Donkin RB, O'Brien T, Maisonobe L, Pietschmann J, Pourbaix D, Steitz P, Worden B, Sadowski G (2011) Commons Math: The Apache Commons Mathematics Library. URL http://commons.apache.org/math/, online; accessed 28/9/2011

Arthur WB (1994) Inductive Reasoning and Bounded Rationality. The American Economic Review 84(2):406–411, DOI 10.2307/2117868, URL http://www.jstor.org/stable/2117868

Axelrod R (1997) The Complexity of Cooperation: Agent-based Models of Competition and Collaboration. Princeton University Press

Banks J, Carson JS (1984) Discrete-Event System Simulation. Prentice Hall

Binko P, Merlino DF, Hoschek W, Johnson T, Pfeiffer A (2004) The CERN Colt library. URL http://acs.lbl.gov/software/colt/, online; accessed 28/9/2011

Bowling M (2005) Convergence and No-Regret in Multiagent Learning. In: Saul LK, Weiss Y, Bottou L (eds) Advances in Neural Information Processing Systems 17, MIT Press, Cambridge, MA, pp 209–216

Bullock S (1997) Evolutionary Simulation Models: On Their Character, and Application to Problems Concerning the Evolution of Natural Signalling Systems. Thesis (phd), University of Sussex

Burstall R (2000) Christopher Strachey: Understanding Programming Languages. Higher-Order and Symbolic Computation 13:51–55

Busoniu L, Babuska R, De Schutt B (2008) A Comprehensive Survey of Multiagent. IEEE Transactions on Systems Man and Cybernetics - Part C: Applications and Reviews 38(2):156–172

Cai K, Gerding E, McBurney P, Niu J, Parsons S, Phelps S (2009) Overview of CAT: A Market Design Competition. Tech. rep., University of Liverpool

Cassell Ba, Wellman MP (2012) Agent-Based Analysis of Asset Pricing under Ambiguous Information: an empirical game-theoretic analysis. Computational & Mathematical Organization Theory DOI 10.1007/s10588-012-9133-y

Chevaleyre Y, Dunne PE, Endriss U, Lang J, Lemâitre M, Maudet N, Padget J, Phelps S, Rodríguez-Aguilar JA, Sousa P (2006) Issues in Multiagent Resource Allocation. Informatica 30:3–31, URL http://www.informatica.si/PDF/30-1/01_Chevaleyre-Issues in Multiagent Resource Allocation.pdf

Chmieliauskas A, Chappin EJL, Dijkema GPJ (2012) Modeling Socio-technical Systems with AgentSpring. In: CESUN Third International Proceedings of the Third International Engineering Systems Symposium: Design and Governance in Engineering Systems - Roots Trunk Blossoms, Delft, Netherlands

Cont R (2001) Empirical properties of asset returns: stylized facts and statistical issues. Quantitative Finance 1(2):223–236, DOI 10.1080/713665670

Cont R, Bouchaud JP (2000) Herd behavior and aggregate fluctuations in financial markets. Macroeconomic Dynamics 4:170–196

Do AL, Rudolf L, Gross T (2010) Patterns of cooperation: fairness and coordination in networks of interacting agents. New Journal of Physics 12(6):063,023, DOI 10.1088/1367-2630/12/6/063023

Dubitzky W, Kurowski K, Schott B (eds) (2011) Repast SC++: A Platform for Large-scale Agent-based Modeling. Wiley, in press

Erev I, Roth AE (1998) Predicting How People Play Games: Reinforcement Learning in Experimental Games with Unique, Mixed Strategy Equilibria. American Economic Review 88(4):848–881

Ficici SG, Pollack JB (1998) Challenges in coevolutionary learning: Arms-race dynamics, open-endedness, and mediocre stable states. In: Proceedings of of ALIFE-6

Ficici SG, Pollack JB (2000) A game-theoretic approach to the simple coevolutionary algorithm. In: Marc Schoenauer, Kalyanmoy Deb, Günter Rudolph, Xin Yao, Evelyne Lutton, Juan Julian Merelo HPS (ed) Parallel Problem Solving from Nature — PPSN VI 6th International Conference, Springer Verlag, Paris, France, URL citeseer.nj.nec.com/322969.html

Ficici SG, Melnik O, Pollack JB (2005) A game-theoretic and dynamical-systems analysis of selection methods in coevolution. DOI 10.1109/TEVC.2005.856203

Fowler M (2004) Inversion of Control Containers and the Dependency Injection pattern. URL http://martinfowler.com/articles/injection.html

Gamma E, Helm R, Johnson R, John Vlissides (1995) Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley

Garrido JM (2001) Object-oriented Discrete-event Simulation with Java: A Practical Introduction (Series in Computer Systems). Kluwer Academic

Gilbert D (2000) JFreeChart. URL http://www.jfree.org/, online; accessed 4/10/2011

Grimmett GR, Stirzaker DR (2001) Probability and Random Processes, 3rd edn. Oxford University Press

Hillis WD (1992) Co-evolving parasites improve simulated evolution as an optimization procedure. In: et al L (ed) Proceedings of ALIFE-2, Addison Wesley, pp 313–324

Iori G, Chiarella C (2002) A Simulation Analysis of the Microstructure of Double Auction Markets. Quantitative Finance 2:346–353

Jackson MO (2007) The Study of Social Networks In Economics. In: Rauch JE (ed) The Missing Links: Formation and Decay of Economic Networks, January, Russell Sage Foundation

Johnson CR, J, Donald K, Sampaleanu C, Arendsen A, Risberg T, Davison D, Kopylenko D, Pollack M, Templier T, Vervaet E, Tung P, Hale B, Colyer A, Lewis J, Fisher M, Brannen S, Laddad

R, Poutsma A, Beams C, Clement A, Syer D, Gierke O, Stoyanchev R (2011) Spring Framework Reference Documentation 3.1. URL http://static.springsource.org/spring/docs/3.1.0.M2/spring-framework-reference/pdf/spring-framework-reference.pdf, online; accessed 27/11/2011

Johnson R, Hoeller J, Arendsen A, Risberg T, Sampaleanu C (2005) Professional Java Development with the Spring Framework. Wiley

Jordan PR, Kiekintveld C, Wellman MP (2007) Empirical Game-Theoretic Analysis of the TAC Supply Chain Game. In: Proceedings of the Sixth International Conference on Autonomous Agents and Multiagent Systems, IFAAMAS, Honolulu, Hawaii, vol 5, pp 1188–1195

Kaisers M, Tuyls K (2010) Frequency adjusted multi-agent Q-learning. In: Van Der Hoek, Kamina, Lespérance, Luck, Sen (eds) Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems, International Foundation for Autonomous Agents and Multiagent Systems, pp 309–316

Krishna V (2002) Auction Theory. Harcourt Publishers Ltd.

Lebaron B (2006) Agent-based Computational Finance. In: Tesfatsion L, Judd KL (eds) Handbook of Agent-Based Computational Economics, vol II, Elsevier

LeBaron B, Yamomoto R (2007) Long-memory in an order-driven market. Physica A: Statistical Mechanics and its Applications 383(1):85–89, DOI 10.1016/j.physa.2007.04.090

Lo A (2005) Reconciling Efficient Markets with Behavioural Finance: The Adaptive Markets Hypothesis. Journal of Investment Consulting 7(2):21–44

Lo AW, MacKinlay AC (2001) A Non-Random Walk Down Wall Street, new ed edn. Princeton University Press

Luke S (2005) MASON: A Multiagent Simulation Environment. Simulation: Transactions of the society for Modeling and Simulation International 82(7):517–527

Matsumoto M, Nishimura T (1998) Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. ACM Transactions on Modeling and Computer Simulation 8(1):3–30

Maynard-Smith J (1973) The Logic of Animal Conflict. Nature 246:15–18

Miller JH (1996) The coevolution of automata in the repeated Prisoner's Dilemma. Journal of Economic Behavior and Organization 29(1):87–112, DOI doi:10.1016/0167-2681(95)00052-6

Minar N, Burkhart R, Langton C, Askenazi M (1996) The Swarm Simulation System: A Toolkit for Building Multi-Agent Simulations. Tech. Rep. 96-06-042, Santa Fe Institute, Santa Fe

Moreira JE, Midkiff SP, Gupta M, Artigas PV, Snir M, Lawrence RD (2000) Java programming for high-performance numerical computing. IBM Systems Journal 39(1):21–56

Nash J (1950) Equilibrium Points in N-Person Games. Proceedings of the National Academy of Sciences of the United States of America 36:48–49

Newman M (2010) Networks: An Introduction. Oxford University Press

Noë R, Hammerstein P (1995) Biological markets. Trends in Ecology and Evolution 10(8):336–339

North MJ, Macal CM (2005) Escaping the Accidents of History: An Overview of Artificial Life Modeling with Repast. In: Artificial Life Models in Software, Springer, chap 6, pp 115–141

Ohtsuki H, Hauert C, Lieberman E, Nowak MA (2006) A simple rule for the evolution of cooperation on graphs and social networks. Nature 441(7092):502–505, DOI 10.1038/nature04605

O'Madadhain J, D F, Nelson T (2011) JUNG - Java Universal Network/Graph Framework. URL http://jung.sourceforge.net, online; accessed 28/9/2011

Palit I, Phelps S (2012) Can a Zero-Intelligence Plus Model Explain the Stylized Facts of Financial Time Series Data? In: Proceedings of the Eleventh International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS) - Volume 2, International Foundation for Autonomous Agents and Multiagent Systems, Valencia, Spain, pp 653–660

Parsons R, MacKenzie J, Fowler M (2000) Plain old Java Object. URL http://www.martinfowler.com/bliki/POJO.html, online; accessed 27/11/2011

Phelps S (2011a) JABM - Java Agent Based Modeling toolkit. URL http://jabm.sourceforge.net, online; accessed 28/9/2011

Phelps S (2011b) JASA - Java Auction Simulator API. URL http://jasa.sourceforge.net, online; accessed 28/9/2011

Phelps S (2012) Emergence of social networks via direct and indirect reciprocity. Journal of Autonomous Agents and Multi-Agent Systems (forthcoming) DOI 10.1007/s10458-012-9207-8

Phelps S, Nevarez G, Howes A (2009) The effect of group size and frequency of encounter on the evolution of cooperation. In: LNCS, Volume 5778, ECAL 2009, Advances in Artificial Life: Darwin meets Von Neumann, Springer, Budapest, pp 37–44, DOI 10.1007/978-3-642-21314-4_5

Phelps S, McBurney P, Parsons S (2010) A Novel Method for Strategy Acquisition and its application to a double-action market game. IEEE Transactions on Systems, Man, and Cybernetics: Part B 40(3):668–674

Prasanna DR (2009) Dependency Injection: With Examples in Java, Ruby, and C#, 1st edn. Manning Publications

Rainsberger JB (2001) Use your singletons wisely
. URL http://www.ibm.com/developerworks/webservices/library/co-single/, online; accessed 27/9/2011

Rayner N, Phelps S, Constantinou N (2011) Testing adaptive expectations models of a double auction market against empirical facts. In: Lecture Notes on Business Information Processing: Agent-Mediated Electronic Commerce and Trading Agent Design, Springer (in press), Barcelona

Rayner N, Phelps S, Constantinou N (2012) Learning is Neither Sufficient Nor Necessary: An Agent-Based Model of Long Memory in Financial Markets. AI Communications (forthcoming)

Richardson C (2006) Untangling Enterprise Java. Queue - Component Technologies 4(5):36–44, DOI http://doi.acm.org/10.1145/1142031.1142045

Santos F, Rodrigues J, Pacheco J (2006) Graph topology plays a determinant role in the evolution of cooperation. Proceedings of the Royal Society B: Biological Sciences 273(1582):51–55, DOI 10.1098/rspb.2005.3272

Shoham Y, Leyton-brown K (2010) Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations, version 1. edn. URL http://www.masfoundations.org/mas.pdf

Tesfatsion L (2002) Agent-based computational economics: growing economies from the bottom up. Artificial Life 8(1):55–82

Vicknair C, Macias M, Zhao Z, Nan X, Chen Y, Wilkins D (2010) A comparison of a graph database and a relational database: a data provenance perspective. In: Proceedings of the 48th Annual Southeast Regional Conference, ACM, New York, NY, USA, ACM SE '10, pp 42:1—-42:6, DOI 10.1145/1900008.1900067

Vriend NJ (2000) An illustration of the essential difference between individual and social learning, and its consequences for computational analyses. Journal of Economic Dynamics and Control 24:1–19

Walsh WE, Das R, Tesauro G, Kephart JO (2002) Analyzing complex strategic interactions in multi-agent games. In: AAAI-02 Workshop on Game Theoretic and Decision Theoretic Agents

Watkins JCH, Dayan P (1992) Q-learning. Machine Learning 8:279–292

Watts DJ, Strogatz SH (1998) Collective dynamics of "small-world" networks. Nature 393(6684):440–442

Weibull JW (1997) Evolutionary Game Theory, First MIT edn. MIT Press

Wellman MP (2006) Methods for empirical game-theoretic analysis. In: Proceedings of the Twenty First National Conference on Artificial Intelligence (AAAI-06), pp 1152–1155

Wilensky U, Rand W (2011) An introduction to agent-based modeling: Modeling natural, social and engineered complex systems with NetLogo. MIT Press, Cambridge, MA, in press

Zimmermann MG, Eguíluz VM, Miguel MS, Spadaro A (2000) Cooperation in an Adaptive Network. Advances in Complex Systems 3(1-4):283–297, DOI 10.1142/S0219525900000212