PRESAGE: A Programming Environment for the Simulation of Agent Societies

Brendan Neville and Jeremy Pitt

Intelligent Systems & Networks Group
Dept. of Electrical and Electronic Engineering
Imperial College London
London, SW7 2BT, UK
brendan.neville@imperial.ac.uk, j.pitt@imperial.ac.uk

Abstract. The paradigm of agent societies has proved particularly apposite for modelling multi-agent systems for networked applications, in particular when the network is open, dynamic and decentralised. In this paper, we describe a software environment which can be used for simulation and animation of these models, allowing a system designer to investigate the complex social behaviour of components, the evolution of network structures, and the adaptation of conventional rules. Effectively, the environment serves as a rapid prototyping tool for agent societies, where the focus of interest is long-term, global system behaviour as much as the verification of specific properties.

1 Introduction

Networked computers and multi-agent systems (MAS) are commonly used as a platform for a new range of applications in for example manufacturing, health, transport, commerce, entertainment, education, and social interaction. Features of these applications include the dynamic network infrastructure, heterogeneous components, unpredicted events, sub-ideal operation (failure to comply to specifications), incomplete and inconsistent information, absence of centralised control, and so on. Techniques from autonomic computing [1] and adaptive systems [2] have proved useful in addressing some of these features; for others the idea of an agent society has been proposed (e.g. [3,4]) which has emphasised the need for conventional rules and social relationships between components.

There still remains though a requirement for system designers and software engineers to retain some understanding of the application under development, and especially of complex systems where random events, erratic behaviour, and self-modification can render the system opaque to mathematical analysis. In the past, rapid prototyping has proved to be an extremely effective tool in helping to understand large-scale MAS, for example in abstracting away from details in order to verify that certain desirable properties hold. However, in autonomic, large-scale MAS, there is an additional requirement not just to verify properties, but also to observe the global outcomes that are the consequence of social interactions and a myriad of independent, local decisions and actions.

In this paper, we propose a rapid prototyping tool whose emphasis is on the simulation of agent societies and the social relationships between agents, allowing the designer to study social behaviour of components, the evolution of network structures, and the adaptation of conventional rules. In this sense, it occupies a space distinct from powerful application development environments, like JADE [5]; agent based modelling and simulation tools [6,7] where the primary purpose is to model and explain the behaviour of non-artificial agents; and other rapid prototyping environments for MAS (e.g. [8,9]) whose principal function is, as stated above, to verify system-wide properties. We illustrate the use of the environment in examining three systems for trust, recommendations, and resource allocation.

The rest of this paper is structured as follows, section 2 provides a set of non-functional requirements and a brief overview of how a user develops and runs a test-bed using the platform. The platform architecture including the underlying simulation model and core modules are discussed in section 3. In section 4 we describe in detail the agent model and agent communication language. Following this section 5 summarises the research which has been carried out using the platform. Finally in 6 and 7 we summarize existing work, conclude and set out our future objectives for the platform.

2 An Overview

To satisfy the functional requirement of developing a rapid prototyping and animation environment for agent societies we have paid particular attention to developing a highly customisable and extensible simulation architecture. However, in order to support the designers goals of observing social behaviour, long term global performance and adaptation we also specifically identify a set of non-functional requirements, namely:

- abstraction: the system allows the designer to tailor the degree of abstraction in their models. In particular, the primary objects of study, the agents and the network, can be as simple or complex as necessary. For example, the agents can range from reactive stubs to fully-fledged BDI agents;
- flexibility: the platform provides many options for parametrisation and reconfiguration. This supports systematic experimentation as the platform can be configured to run with the independent variables set over a range of values, and the measures of interest (dependent variables) collected for each run;
- extensibility: the platform is provided with a pre-programmed set of libraries, but the designer may extend the functionality using scriptable methods and component plug-ins;
- interaction: particular emphasis is given to simplifying the front-end to 'program' an experiment, to visualise the animation as it is running, data logging, and access to external applications, such as Gnuplot, for graphical representation of data;
- scalability: the architecture of the system has been designed to support both single-processor and distributed animation, allowing simulation to feature societies comprising many hundreds of agents.

In developing a prototype the experimenter can create their agent participant types through optional use of the supplied abstract class; to ensure compatibility with the simulation calls and provide core functionality like message handling etc. They can then choose from one of the pre-defined network and physical environment modules or extend the basic Network and PhysicalWorld classes to suit their purpose. Finally they may add functionality to the platform in the form of scriptable methods and plugins.

A basic input-output overview of our simulation platform is illustrated by Fig. 1. The experimenter configures each simulation run via input-files; these files serve four main purposes, parametrising the general simulation variables, configuring the simulated agents (participants), scripting events and initialising the required plug-ins.

Once the platform has initialised as specified it enters the simulation thread and loops for the required number of iterations. During this time the user can view the progress of the simulation via visualiser plug-ins, record data using data archiver plug-ins, execute methods and launch extra plug-ins during runtime.

At the end of the simulation, the platform can be scripted to organise and archive results and input-files. It can also call external applications for example Gnuplot to create publication ready graphs.

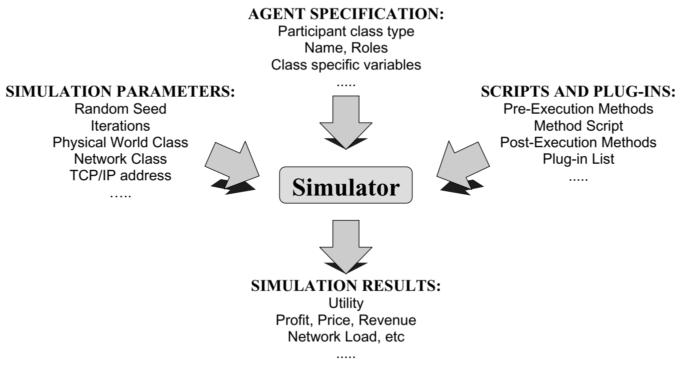

Fig. 1. Input-Output Overview of the Simulation Platform

3 Platform Architecture

The PRESAGE architecture is illustrated as a software stack (Fig. 2) depicting the base simulation module, the interfaces and abstract classes, simulation managers, and the platforms connectivity to external processes. Above this we have given some examples of how the user could utilise the classes and modules e.g. an auction scenario operating over a unstructured P2P network without a physical world. In the following sections we address each of the modules in more detail.

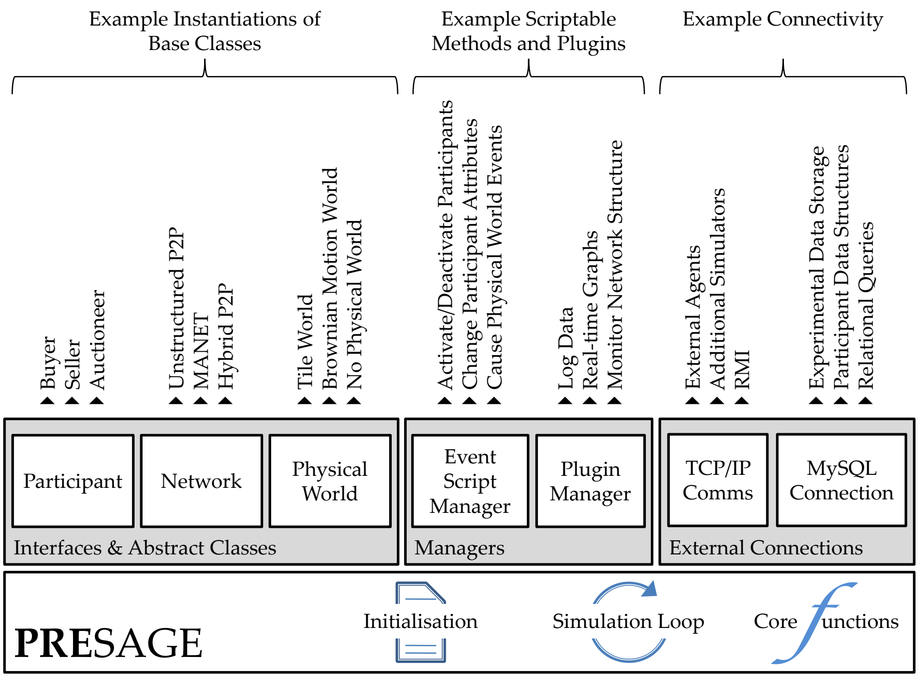

Fig. 2. Representation of the Architecture of the platform

3.1 The Base Simulation Module

The role of the base simulation module is to perform parameter initialisation, manage simulation execution, and provide generic functions to higher level modules and classes. PRESAGE takes a multi-agent discrete time-driven approach. In this simulation execution model each loop of the simulation control thread equates to a simulation time slice. For every time slice the state of the network and physical world is updated, scripted events execute, plugins perform their duties and the agent participants are given a turn to perform physical and communicative actions. By handling the agent process execution as a centralised time-driven model we ensure pseudo-concurrent execution of agent actions thus affording the advantages of Multi-agent based simulation (MABS) outlined in [10], and providing the user and agent a centralised notion of time. Concurrency is enforced by queuing actions until the end of each time slice.

We have developed a time-driven as opposed to an event-driven model of execution because:

– While event-driven models are generally seen as more computationally efficient than time-driven models due the former' ability to overlook periods of inactivity, such efficiency is absent in the case of simulating agents, since they react to changing conditions and are therefore required to constantly sense their environment.

- The complexity of programming discrete-event models increases rapidly with the complexity and heterogeneity of the agents and the number of event types. Whereas in a time-driven model the agents may become more complex, however, the interface between the agent and the simulation model does not.
- In event-driven models, the simulator determines in advance the next event based on the current state of the world and steps directly to it (without animating the states of the world in-between). This is inappropriate for our simulation execution model as we require it to be indifferent to participant architecture and facilitate probabilistic behaviour (for Monte Carlo experiments), pro-activity and adaptability.

3.2 Managers

This section introduces the three simulation managers which afford control over the simulation execution, plugins and the execution of extraneous events. The simplest of these is the Control Panel. This primarily lets the user run/pause and step through the simulation. In addition to providing progress information and allowing the user to prematurely end a simulation whist still executing post processing, archiving and tidying up of the databases and connections.

The Event Script Manager (ESM) uses the Java reflection API to facilitate runtime execution of Java methods. This allows the user to script the execution of a method at a certain time point with specific variables independent of the platforms compile time behaviour. This script initially takes the form of an input file, but events can be added through the GUI during runtime. Methods can also be scripted for execution before or after the simulation run such that the user can use them for initialisation or post processing. Given the generic nature of scripting method execution there are a vast array of possible uses, these include, triggering events in the simulated physical world, adapting the network topology, altering parameters and timing each agent's entrance or exit from the simulation.

The Plugin Manager (PM) allows the user to launch plugin modules from input files or a GUI during runtime; the key difference between plugins and methods being that plugins persist between simulation cycles meaning that they are repeatedly executed, have memory between simulation cycles and can include a user interface. As a result they form the basis of the many possible data archiving and visualisation tools. The PM, like the ESM auto-detects available plugins and allows the user to launch and remove them during runtime. A plugin can be created by simply using the `plugin` interface. The power of the plugin architecture is illustrated by two key plugins, the DataArchiver and the Visualiser.

DataArchiver: One key feature of any simulation platform is the ability to log experimental data. A basic DataArchiver plugin class is provided in the platform API that can create results logs in the form of spreadsheets. The specific nature of the data and its layout in the output-file is defined by the user as it is scenario dependant. This is relatively easy process of instantiating the DataArchiver's abstract method `getDataRow()` to return a row of data in the form of an array. In each simulation cycle the plugin will then get the required data and archive it in a comma separated file.

Visualisation Plugins: We provide a small group of plugins specifically designed to enable the user to create realtime visualisations of experimental data. At this time we have created three basic forms: line graphs, radial plots and network visualisation. While it is our intention to extend this library further in the future, the user can, of course, create their own as needed.

3.3 External Connections

The platform supports many types of external connection. In this section, we review three, TCP/IP connections, MySQL, and access to other external applications.

TCP/IP Communication consists of a client/server pair for communicating with external processes such as situated agents, remote servers and networking the platform to additional simulators.

The MySQL connection is managed by the platform for providing short-cut methods to perform queries and updates, in addition to managing the java-sql connection (jdbc). This enables users to store large volumes of simulation data including event logs for post-processing. The participants can also use SQL to: store beliefs, form temporary data structures from more than one table, perform mathematical functions on large datasets or quickly and efficiently search and organise a large amount of information.

External Process Invocation is handled by a number of convenience methods allowing the execution of system commands and external applications from within the platform. These can either be called by user defined code in the network, physical world, participant, or plugin classes; or by a scripted event. This is particularly useful for launching agent processes outside the simulation, calling on Gnuplot or a spreadsheet application to post-process simulation results.

3.4 Environmental Interfaces and Abstract Classes

Agent systems operate in a number of physical and network environments from fully connected static networks without the need to model a physical world to vehicular adhoc networks (VANETS). The individual properties of these environments pose unique challenges to the agent system developer, therefore it is essential that agent simulation platforms support the custom specification of these environments. In order to achieve this the PRESAGE platform contains two abstract classes namely the Network Simulation Module and a Physical World Simulation Module.

Network Simulation Module: The network module's core function is to facilitate the exchange of messages between connected peers and to simulate dynamic connectivity between the participants. Network modules are simple to create by extending the basic abstract class to determine the required behaviour. The

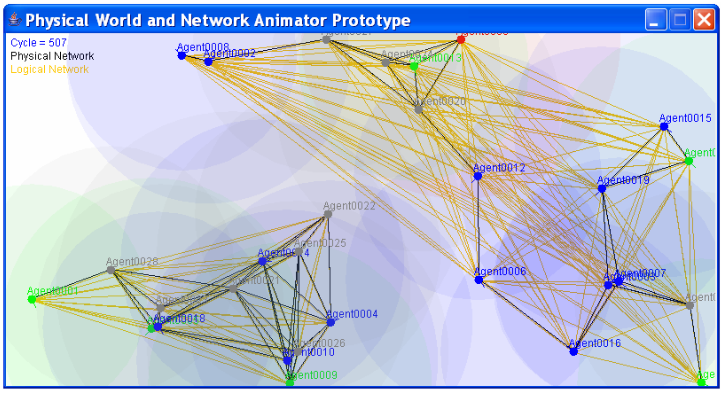

Fig. 3. Plugin creates a realtime animation of the changing positions of the agents in the simulated physical world and the effect this has on the topology of the physical and logical networks

following network types have been created: static fully/partially connected, unstructured P2P, hybrid P2P and mobile adhoc networks.

Physical World Simulation Module: The platform supports the inclusion of a simulated spatial environment for the agents. Like the simulated network, the physical world is an interface class which allows for custom specification by the experimenter. The basic interface supports the addition and removal of participants from the world and facilitates the sensing and effecting of their environment. It is up to the user to define the valid actions and their effect on the state of the world, in addition to any rules of the environment not determined by agent behaviour.

An example of using the physical world and network interfaces is an instance of a wireless mobile adhoc network (MANET) simulation. In our configuration the agents can move in a two dimensional environment and can sense the relative position of nearby peers. The world itself bounces agents when they interact with its boundaries causing the agents to move in a Brownian motion. The spatial data of the world is used by the network module to calculate the physical connections of the network based on relative distances between peers and their wireless transmission ranges. The physical network topology can then be used to infer the logical connectivity of the participants. A realtime visualisation of the physical world and the resultant network is provided by a plugin shown in Fig. 3.

4 Agents, Participants and Communication

The principal component of the platform is its collection of agents, whose interaction with one another and their environment is our primarily interest. In theory it would be ideal for the platform not to constrain the design of the agents

Table 1. Required methods and variables for a simulation participant

Variables		
public String	gUID	globally unique identifier: defined from input file
public Queue	inbox	to allow the network module to enqueue messages to the agent
Methods		
public boolean	isRole(String role);	returns true if role is one of the participants roles.
public void	execute();	called by the simulation thread upon a participants turn.
public void	onActivation();	called by the platform when the agent becomes active in the simulation.
public void	onDeActivation();	called by the platform when the agent is removed from the simulation.

in any way. However in order to interact with the base simulation model and ensure the interoperability of participants a degree of homogeneity is required. Table 1 lists these prerequisites. Externally the agent must have a globally unique identifier (GUID), defined roles and communicate via a common agent communication interface (as defined in the following section). However, internally the requirements simply facilitate the interaction with the simulation platform, for instance activation/deactivation of the agent and calling the agent to take it's turn via a public methods e.g. `execute()`. The user may also customise the simulation thread to allow them to interleave the execution of agents, this is achieved by replacing the `execute()` method with a series of sub-methods. This is the approach used in the example applications in section 5. Within these constraints the user is free to develop their own agent architecture be it reactive, deliberative, BDI or otherwise. As such the platform is neutral with respect to the agents' internal architectures.

4.1 The Participant Class

It is expected that the majority of users of the platform will be primarily interested in the interaction between agents and the evolution of behaviour within a simulated agent society. As such we have developed a root agent class `Participant` from which researchers can derive heterogeneous agents for participation in their simulations. Figure 4 shows how one might derive the necessary classes for an online auction scenario in a Virtual Organization and instantiate an heterogeneous population from them. Notice that the class hierarchy allows us to define more or less sophisticated agent strategies: from the simple buyer, socio-cognitive buyer, and onto machine learning or game theoretic buyers. The participant class handles as much of the agent's internal operation as possible

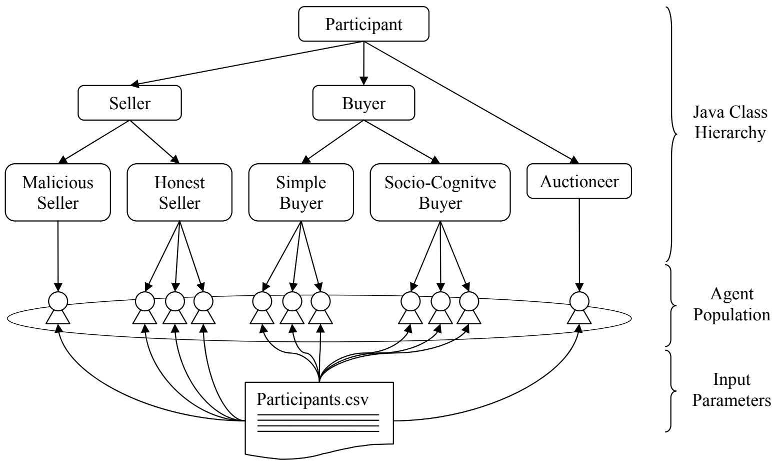

Fig. 4. Using Java OOP, Participant class and input files to define a heterogeneous agent population

(without sacrificing scenario flexibility). Its architecture is a combination of deliberative and probabilistic models, this has proved sufficiently complex for our experiments in emergent behaviour.

To create the individual participants The derived classes must then be launched and parametrised via an input-file. Each row of the file launches and specifies an individual agent's parameters. The core inputs the user must specify for each agent include, among other things, the Java class that includes the agents reasoning and communication protocols which extend our generic participant class, the agents globally unique identifier, the initial roles to be assigned to the agent. In addition the user can provide scenario specific parameters for example in our trust and e-commerce scenario the participants input file also defines what trust model each agent will adopt, its economic constraints/preferences and its character type e.g. its inclination towards and strategy for illegal, unethical and antisocial behaviour.

4.2 Agent Communication

The simulation platform aims to put minimal restriction on the internal characteristics of the participating agents. However, in order for the agents to communicate effectively some *a priori* knowledge as to the mechanisms and semantics of communication are required. Following, Pitt and Mamdani [11] who argue for the use of a protocol based semantics in the external specification of agent interaction specifically between agents with behavioural and architectural heterogeneity. Within the participant class we provide the necessary mechanisms for handling protocol based communication between the agents. In fact all the mechanisms from message sending and parsing to maintaining the state of current interactions is built in; effectively reducing the users work load to defining the protocol and the reasoning of the agents at each stage of that protocol.

In this section we discuss the defined agent communication interface which permits and facilitates the exchange of information between peers. The interface consists of a higher and lower level component pair: the agent communication language (ACL) and the mechanism for transmitting messages. Message transmission is achieved by calling the

```
sendMessage(Message, InetAddress)
```

method of the Network module. The Network module will either send the message via TCP/IP sockets or enqueue the message to the recipient's inbox queue; depending on whether the recipient is internal or external to the platform. We define our ACL in terms of three components: The message syntax, the mechanisms maintaining the state of a communicative context (a conversation) and the external semantics of the protocols. The following three subsections discuss the way that messages, conversations and protocols are represented, in order to give the user an understanding of how to implement a protocol and associated agent behaviour within our framework.

Message Syntax. In order for agents to parse and interpret information exchanged between them there must be an agreed upon message syntax. In our ACL a message takes the form of a seven-place term (see below) where the terms R and S denote the intended recipient and the sender respectively; these are instantiated with the agents $GUID$ values. Element C defines the type of communicative act (i.e query or purchase) being performed. P determines the protocol (i.e. CNP or Hello) under which the communicative act is being issued. CKs and CKr are the conversation keys ($ConvKey$) of the sender and recipient respectively; these are used by the agents to recognise the ongoing context in which a message should be interpreted (Pitt and Mamdani [11]). When an agent initiates a conversation they create a conversation object and a instantiate it with a locally unique conversation key. This key is then sent with all subsequent messages. When an agent receives a message without an instantiated CKr it signals that this is the first message of a new conversation; the recipient will then create a new conversation and instantiate its key before processing the message. The format of the message contents is determined by the message performative C and the protocol P being followed.

$$\text{message}\,(R, S, C, P, CKr, CKs, Content)\,;$$

$$\begin{aligned}&\text{message (Agent0056, Agent0022, introduction, hello,}\\&\qquad\text{(.), (4.0), contents(Agent0022}, \langle\text{consumer}, 127.0.0.1 : 9436\rangle));\end{aligned}$$

Conversations. As an agent executes an interaction protocol with a peer it must maintain local information about the context of that interaction. The agents achieve this by creating a *conversation* object for every multi-stage interaction initiated. A conversation object has the following structure:

$$\text{conversation}\,(CKm, CKt, tID, P, S, To, Beliefs)\,;$$

CKm and CKt are the agents ConvKey and its peer's respectively. These ConvKeys are used to link incoming messages to an ongoing conversation and to instantiate the ConvKey fields of any replies. The fields tID and P identify whom the conversation is with and which protocol they are following. The state of the conversation S identifies at which point of the protocol (and therefore which section of the agents reasoning) the next message or timeout should refer to. To is the time at which the conversation is internally called, this can happen for a number of reasons: it could be used to end a period of open bidding in an auction protocol or simply to call a conversation to resend a message or tidy up if a peer has failed to respond. Finally the beliefs field is a set of temporary beliefs which the agent wishes to directly associate with a conversation, for instance the current highest bid in an auction.

It is necessary that the participant are able to carry out multiple conversations at any given time; the set of active conversation objects are stored in the conversations KB. Periodically the agent checks to see if any of the conversations have timed out or have completed. If the state of a conversation is completed then the conversation is removed from the KB. However if the conversation has timed out: the code associated with the protocol is passed the conversation. When we refer to the code associated with a conversation, we are referring to the user defined method that defines the agents behaviour at each stage of the protocol as describe in the next section.

Protocols and User Defined Semantics. The Participant class uses the Java reflection API in order to provide a user extensible protocol library. To add a protocol to the agent the user simply creates two methods:

```
protected void protocol_name(Message msg)

protected void protocol_name(Message msg, ConvKey convkey)
```

The first method is called on receipt of any message claiming to conform to the protocol. This method performs a number of checks before calling the second method; for instance if the message is part of an ongoing conversation and if that conversation actually exists; or if the message is intended to start a new conversation in which case it will create a new conversation object. The second method is called in three situations: agent receiving a message (via the first method), a conversation timeout in which case the message is null and finally as a result of a child conversation returning. It is in the second method that the user codes the relevant agent behaviours for each stage of the protocol.

This handling of messages and conversations is added to the Participant class for the convenience of users whom do not require a specific agent architecture. With more advanced applications users can override built in conversation and messaging functions allowing messages to instead be passed over to code written in languages supported by the Java Native Interface including among others Prolog, C++ and Smalltalk.

5 Sample Applications

In this section we summarise three agent systems which we have prototyped and simulated within the PRESAGE platform.

5.1 Social Networks and Recommendation

In this scenario the prototype system under investigation is a P2P recommendation network. Whereby differing peer preferences gives rise to states of inconsistent union and the distributed architecture results in peers maintaining local, subjective and incomplete recommendation sets. The aim is for agents to base their purchase decisions on the recommendations of peers with similar preference. To do this in a traditional centralised collaborative filtering system, a server models the degree of similarity between all the peers based complete knowledge of their opinions and then uses this data to infer a desirability score for each agent to untried content pairing. Within a distributed environment this approach would cause significant computational scalability and network loading issues. We are using PRESAGE to simulate an agent society whereby peers model one another based on only the locally available recommendations. By using these peer models to self-organise their network connections the agents can exploit the localised and incomplete nature of the network to pre-filter recommendations thus increasing the utility of the incoming recommendations without being required to compute models for every peer, discovering all the available products or replicating their beliefs across the system.

5.2 Open Distributed Agent Mediated Marketplace

In this study we investigated the behaviour of an agent mediated marketplace which was intrinsically unmoderated, dynamic, and which could not guarantee that its participants would behave honestly, ethically and competently. The agents were adapted by integrating a framework for socio-cognitive reasoning (trust, recommendation and reputation) into the individual agents economic decision making. The results of our simulations show that the integration of social behaviour into the trading agent architecture can not only act as an effective mechanism for discouraging norm-violation, but also minimise the detrimental economic inefficiencies resulting from the protective measures. Details of this work can be found in [12,13].

5.3 Adaptation of Voting Rules

In previous work with agent-based mobile ad-hoc networks, vehicular networks, and virtual organizations a common scenario is for the collective use of a limited common resource. In this application of the platform, we have defined a multi-agent system which is highly volatile, in the sense that agents can be (unpredictably) ‘present’ or ‘absent’ in any time slice. The agents that are present have to vote on the distribution of resources. The problems are, firstly, to decide on an ‘equitable’ distribution of resources without depleting it (i.e. the ‘tragedy of the commons’); and secondly, to adapt the voting rules in one time slice to (try to) ensure a ‘safe’ allocation in the next one. Presage is proving useful here because, given the range of different possible characteristics and complex functionality of the agents, it is straightforward to generate and configure a large and diverse population mix. The time-driven execution model supports the time slice allocation of resources, and the visualisation allows us to follow, at run-time, the key dependent variables as they change: network structure, voting rules, agent ‘satisfaction’, and resource allocation. Details can be found in [14].

6 Related Work

Rapid prototyping and animation of agent societies in a logical form has been effectively used in order to demonstrate and verify properties of agent societies. In [8] Vasconcelos et al present an approach to rapid prototyping multi-agent systems through the definition of a global interaction protocol. The global protocol defines the types and order of interaction between the components is used to automatically generate a set of agents which are simulated to check for desirable properties in the protocol. CaseLP [9] is a logic-based prototyping environment for specifying and verifying complex distributed applications it also provides tools for specifying certain network properties when developing prototypes of distributed systems e.g. reliability or latency of connections. We argue that these approaches provide a complimentary perspective to the one offered by PRESAGE where we are primarily interested in the global long run outcomes and dynamic behaviour of the system. PRESAGE can however produce a narrative (a sequence of actions) of its simulation of the society, this combined with a description of the social states can be used to invoke tools such as the Society Visualiser[3] to check system or protocol properties.

Muli-agent Based Simulation (MABS)[10] is a micro-level approach to simulation of complex systems. Whereby the behaviour of system components or individuals are modelled as agents. A number of MABS tools exist [15] including Swarm[1][6], Repast [2] and MASON [7] and are widely used for Agent Based Social Simulation (ABSS)[16,17,18]. In contrast to MABS the PRESAGE platform is intended as a multi-agent based simulator for agent societies; as opposed to MABS where generally the focus is upon modelling a non-agent system as a system of agents. Hence our requirement for supporting heterogeneous agent architectures and simulating properties of communication networks. ABSS uses MABS techniques to model human interactions within a multi-agent system, generally with relatively simple behaviours (on an individual agent basis) that when simulated their interactions lead to complex global behaviour. These results are subsequently used to understand and elaborate social theories. Our social agent experiments [12,13] cross-fertilise with the theories and formalisations of the ABSS field and the wider social sciences. However our interest lies

[1] www.swarm.org

[2] http://repast.sourceforge.net/

in using this knowledge to solve problems related to open agent societies which diverges from their use to further understand human society.

Multi-agent system development tools (for an evaluation see [19]) such as AgentBuilder[3] and JADE support analysis, design, development and deployment of multi-agent applications. More specifically, JADE[4] [5,20,21] is a robust middleware for developing FIPA[5] compliant agent applications. The JADE agent framework provides the developer with an API for message syntax and parsing and a set of standard interaction protocols thus simplifying the process of developing interoperable agents. There also exists a deployment tool [22] which supports the configuration and deployment of JADE agent based applications.

We view PRESAGE prototyping as a step before frameworks like JADE or AgentBuilder; providing a platform for investigating system wide performance, emergent behaviour, optimising interaction protocols and algorithms. We are currently working on supporting a deployment path for our test-bed participants: currently users are able to wrap the simulation participants in a class that allows them to function independently of the simulation platform.

One element of our future work is a thorough analytic comparison of PRESAGE functionality with respect to other Multi-agent programming tools such as MABS, JADE, AgentBulider, etc. Based on criteria such as, for example the type of agent, the type of society, its intended use, its intended users and so on.

7 Summary and Conclusions

Given the complex and dynamic nature of agent societies including self-governance, evolving norms and emergent behaviour, the process of developing systems which robustly exhibit desirable system wide behaviours under conditions which cannot be guaranteed; can be time-consuming and complex. In this paper we have described our approach to rapid prototyping and testing agent societies. PRESAGE affords the user centralised global monitoring and simulation control, is flexible and extensible through the use of abstract classes, event-scripting and plugins; supports heterogeneous agent architectures and the simulation of an agent system's underlying dynamic network architecture.

We have given examples of our platform being utilised to prototype and test open distributed agent systems featuring proactive behaviour, lack of centralised control, heterogeneity and adaptability. Further more our institution is currently utilising the PRESAGE platform to investigate application of game theory, alternative dispute resolution, opinion formation and norm emergence in agent societies.

Through this experience we intend to fully document and refine the framework before contributing it to the wider community. We designed and built the platforms's functional architecture to support the non-functional specification

[3] www.agentbuilder.com
[4] http://jade.tilab.com
[5] www.fipa.org

presented in section 2, however, as this is an ongoing development we have yet to implement these requirements in full. Future work includes tools for automated exploration of the parameter space, greater variety of predefined agent architectures and extended support for deployment on multi-core and cluster computing.

Acknowledgments

This work has been undertaken in the context of the EU-funded ALIS[6] project(IST 027968). We are also grateful for the useful comments of the anonymous referees.

References

1. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1), 41–50 (2003)
2. DeLoach, S., Oyenan, W., Matson, E.: A capabilities-based model for adaptive organizations. Autonomous Agents and Multi-Agent Systems 16(1), 13–56 (2008)
3. Artikis, A., Pitt, J., Sergot, M.: Animated specifications of computational societies. In: Castelfranchi, C., Johnson, L. (eds.) Proc. of the First International Conference on Autonomous Agents and Multi-Agent Systems, pp. 1053–1062. ACM Press, New York (2002)
4. Sierra, C., Rodríguez-Aguilar, J., Noriega, P., Esteva, M., Arcos, J.: Engineering multi-agent systems as electronic institutions. European Journal for the Informatics Professional V(4), 33–39 (2004)
5. Bellifemine, F., Poggi, A., Rimassa, G.: Jade - a fipa-compliant agent framework. In: Proceedings of PAAM 1999, pp. 97–108 (April 1999)
6. Minar, N., Burkhart, R., Langton, C., Askenazi, M.: The swarm simulation system, a toolkit for building multi-agent simulations (1996)
7. Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., Balan, G.: Mason: A multiagent simulation environment. simulation 81(7), 517–527 (2005)
8. Vasconcelos, W., Robertson, D., Sierra, C., Esteva, M., Sabater, J., Wooldridge, M.: Rapid prototyping of large multi-agent systems through logic programming. Annals of Mathematics and Artificial Intelligence 41(2-4), 135–169 (2004)
9. Martelli, M., Mascardi, V., Zini, F.: A logic programming framework for componentbased software prototyping (1999)
10. Davidsson, P.: Multi agent based simulation: Beyond social simulation. In: Moss, S., Davidsson, P. (eds.) MABS 2000. LNCS, vol. 1979, pp. 97–107. Springer, Heidelberg (2001)
11. Pitt, J., Mamdani, A.: A protocol-based semantics for an agent communication language. In: Proceedings 16th International Joint Conference on Artificial Intelligence IJCAI 1999, pp. 485–491. Morgan-Kaufmann, San Francisco (1999)
12. Neville, B., Pitt, J.: A computational framework for social agents in agent mediated e-commerce. In: Omicini, A., Petta, P., Pitt, J. (eds.) Engineering Societies in the Agents World IV. Springer, Heidelberg (2004)

[6] www.alisproject.eu

13. Neville, B., Pitt, J.: A simulation study of social agents in agent mediated e-commerce. In: Proceedings of the Seventh International Workshop on Trust in Agent Societies (2004)
14. Carr, H., Pitt, J.: Adaptation of voting rules in agent societies. In: Proceedings AAMAS Workshop on Organised Adaptation in Multi-Agent Systems (OAMAS) (2008)
15. Gilbert, N., Bankes, S.: Platforms and methods for agent-based modeling. Proc. of the National Academy of Sciences of the United States of America 99(10), 7197–7198 (2002)
16. Conte, R., Edmonds, B., Moss, S., Sawyer, R.K.: Sociology and social theory in agent based social simulation: A symposium. Comput. Math. Organ. Theory 7(3), 183–205 (2001)
17. Conte, R.: Agent-based modeling for understanding social intelligence. Proceedings of the National Academy of Sciences of the United States of America 99(10), 7189–7190 (2002)
18. Davidsson, P.: Agent based social simulation: A computer science view. J. Artificial Societies and Social Simulation 5(1) (2002)
19. Ricordel, P.M., Demazeau, Y.: From analysis to deployment: A multi-agent platform survey. In: Omicini, A., Tolksdorf, R., Zambonelli, F. (eds.) ESAW 2000. LNCS, vol. 1972, pp. 93–105. Springer, Heidelberg (2000)
20. Bellifemine, F., Poggi, A., Rimassa, G.: Developing multi-agent systems with jade. In: Castelfranchi, C., Lespérance, Y. (eds.) ATAL 2000. LNCS, vol. 1986, pp. 89–103. Springer, Heidelberg (2001)
21. Bellifemine, F., Rimassa, G.: Developing multi-agent systems with a fipa-compliant agent framework. Softw. Pract. Exper. 31(2), 103–128 (2001)
22. Braubach, L., Pokahr, A., Bade, D., Krempels, K.-H., Lamersdorf, W.: Deployment of distributed multi-agent systems. In: Gleizes, M.-P., Omicini, A., Zambonelli, F. (eds.) ESAW 2004. LNCS, vol. 3451, pp. 261–276. Springer, Heidelberg (2005)