The Second International Workshop on

INFRASTRUCTURES AND
TOOLS FOR MULTIAGENT
SYSTEMS

ITMAS 2011

May 2nd, 2011
Taipei, Taiwan

Joint with the Tenth International Conference on
Autonomous Agents and Multiagent Systems
(AAMAS 2011)

Vicent Botti
Ana Garcia-Fornés
Michal Pechouček
Alessandro Ricci
Jose M. Such
Danny Weyns
Workshop Organization

Programme Chairs

Vicent Botti (Universitat Politècnica de València, Spain)
Ana Garcia-Fornes (Universitat Politècnica de València, Spain)
Michal Pechouček (Czech Technical University in Prague, Czech Republic)
Alessandro Ricci (Alma Mater Studiorum-Università di Bologna, Italy)
Jose M. Such (Universitat Politècnica de València, Spain)
Danny Weyns (Katholieke Universiteit Leuven, Belgium)

Programme Committee

Juan M. Alberola (Universitat Politècnica de València, Spain)
Matteo Baldoni (Università degli Studi di Torino, Italy)
Fabio Bellifemine (Telecom Italia, Italy)
Juan A. Botía (University of Murcia, Spain)
Vicent Bottí (Universitat Politècnica de València, Spain)
Juan M. Corchado (University of Salamanca, Spain)
Yves Demazeau (Laboratoire d’Informatique de Grenoble, France)
Nadia Erdogan (İstanbul Teknik Üniversitesi, Turkey)
Agustin Espinosa (Universitat Politècnica de València, Spain)
Marc Esteva (IIIA-CSIC, Spain)
Ana García-Fornes (Universitat Politècnica de València, Spain)
Dominic Greenwood (Whitestein Technologies, Switzerland)
Jomi F. Hübner (Federal University of Santa Catarina, Brazil)
Kamalakar Karlapalem (Int. Institute of Information Technology, India)
Yasuhiiko Kitamura (Kwansei Gakuin University, Japan)
Abder Koukam (University of Technology UTBM, France)
Michal Laclavík (Slovak Academy of Sciences, Slovak Republic)
Tim Miller (University of Melbourne, Australia)
Pavlos Moraitis (Paris Descartes University, France)
Andrea Omicini (Alma Mater Studiorum-Università di Bologna, Italy)
Sascha Ossowski (University Rey Juan Carlos, Spain)
Julian Padget (University of Bath, UK)
Michal Pecheouček (Agent Technology Center, Czech Republic)
Alessandro Ricci (Alma Mater Studiorum-Università di Bologna, Italy)
Juan A. Rodríguez-Aguilar (IIIA-CSIC, Spain)
Murat Şensoy (University of Aberdeen, UK)
Carles Sierra (IIIA-CSIC, Spain)
Michael Shumacher (University of Applied Sciences Western, Switzerland)
Jose M. Such (Universitat Politècnica de València, Spain)
Pavel Vrba (Rockwell Automation Research Center, Czech Republic) 
Danny Weyns (Katholieke Universiteit Leuven, Belgium)

External Reviewers

María Del Carmen Delgado (IIIA-CSIC, Spain)  
Stéphane Galland (University of Technology UTBM, France)  
Moser Silva Fagundes (University Rey Juan Carlos, Spain)  
Nikolaos Spanoudakis (Technical University of Crete, Greece)
Preface

ITMAS 2011 follows the success of its predecessor ITMAS 2010, which was the very first edition of ITMAS. ITMAS 2010 was held in conjunction with AAMAS 2010 in Toronto (Canada). ITMAS 2011 is again to be held in conjunction with AAMAS 2011, this time in Taipei (Taiwan). This year we had 17 submissions from which 9 were finally accepted. This confirms both the relevance and interest of the workshop. Moreover, all of the submissions received were of high quality.

ITMAS aims at bringing together leading researchers from both academia and industry to discuss issues on the design and implementation of infrastructures and tools for Multiagent Systems. When developing applications based on Multiagent Systems, developers and users demand infrastructures and tools which support essential features in Multiagent Systems (such as agent organizations, mobility, etc.) and facilitate the system design, management, execution and evaluation. Agent infrastructures are usually built using other technologies such as grid systems, service-oriented architectures, P2P networks, etc. In this sense, the integration and interoperability of such technologies in Multiagent Systems is also a challenging issue in the area of both tools and infrastructures for Multiagent Systems. A long term goal is the industrial development of infrastructures for building highly scalable applications comprising pre-existing agents that must be organized or orchestrated.

In order for Multiagent Systems to be included in real domains such as media and Internet, logistics, e-commerce and health care, infrastructures and tools for Multiagent Systems should provide efficiency, scalability, security, management, monitorization and other features related to building real applications.

Alessandro, Anna, Danny, Jose, Michal, and Vicent.
# Table of Contents

Agent-based Simulation Platform Evaluation in the Context of Human Behavior Modeling ......................................................... 1  
Michal Laclavík, Štefan Dlugolinský, Martin Šeleng, Marcel Kvas-say, Bernhard Schneider, Holger Brucker, Michal Wrzeszcz, Jacek Kitowski, Ladislav Hluchy

An Agent Infrastructure for Privacy-enhancing Agent-based E-commerce Applications ......................................................... 16  
Jose M. Such, Agustin Espinosa, Ana Garcia-Fornes

Auto-adaptation of Open MAS through On-line Modifications of the Environment ................................................................. 31  
Roberto Centeno, Holger Billhardt

Combining Semantic Web and Logic Programming for Agent Reasoning ................................................................. 46  
Murat Şensoy, Wamberto W. Vasconcelos, and Timothy J. Norman

Cost-Aware Reorganization Service for Multiagent Systems ........ 60  
Juan M. Alberola, Vicente Julian, Ana Garcia-Fornes

Enforcing Norms in Open MAS ....................................................... 75  
Natalia Criado, Estefania Argente, Pablo Noriega, Vicent Botti

Evolving Semantics for Agent-based Collaborative Search ........ 90  
Murat Şensoy

Micro-agents on Android: Interfacing Agents with Mobile Applications 105  
Christopher Frantz, Mariusz Nowostawski, Martin Purvis

Prognostic agent assistance for norm-compliant coalition planning . . 120  
Jean Oh, Felipe Meneguzzi, Katia Sycara, and Timothy J. Norman
Agent-based Simulation Platform Evaluation in the Context of Human Behavior Modeling

Michal Laclavík¹, Štefan Dlugolinský¹, Martin Šeleng¹, Marcel Kvassay¹, Bernhard Schneider², Holger Bracker², Michał Wrzeszcz³, Jacek Kitowski³, Ladislav Hluchý¹

¹Institute of Informatics, Slovak Academy of Sciences, Dúbravská cesta 9, 845 07 Bratislava, Slovakia
{laclovik.ui, stefan.dlugolinsky, martin.seleng, marcel.kvassay, hluchy.ui}@savba.sk
²EADS Deutschland GmbH
Landshuter Straße 26, 85716 Unterschleißheim, Germany
{bernhard.schneider, holger.bracker}@cassidian.com
³Academic Computer Centre CYFRONET,
University of Science and Technology in Cracow, Poland
michalwrzeszcz@gmail.com, kito@agh.edu.pl

Abstract. In this paper we provide a brief survey of agent based simulation (ABS) platforms and evaluate two of them – NetLogo and MASON – by implementing an exemplary scenario in the context of human behavior modeling. We define twelve evaluation points, which we discuss for both of the evaluated systems. The purpose of our evaluation is to identify the best ABS platform for parametric studies (data farming) of human behavior, but we intend to use the system also for training purposes. That is why we also discuss one of serious game platform representatives – VBS2.

Keywords: agent-based simulation, human behavior modeling.

1 Introduction

Human Behavior Modeling is an important area of computational science with implications not only for social sciences, but also for economics, epidemiology and other fields. Scientific literature abounds in heterogeneous and highly specialized, theoretically founded concepts of human cognition, emotion and other behavior aspects. The task to find a simulation framework that would allow effective implementation of such conceptions for different aspects of real human behavior to interoperate is particularly challenging. Our motivation for this paper derives from the EDA project A-0938-RT-GCEUSAS (European Urban Simulation for Asymmetric Scenarios) whose goals and requirements provide the context and a guideline for our evaluation of the existing systems.

The EUSAS project focuses on asymmetric security threats in urban terrain. Its goal is to develop an all-in-one tool enhancing the mission analysis capabilities as
well as virtual training of real human beings (security forces) in a highly realistic 3D cyber environment. In virtual trainings, simulated characters (civilians) with highly realistic patterns of behavior would interact with real people (security forces), while in the mission analysis (Data Farming) mode both the civilians and the security forces would be simulated. A natural choice for the simulations of this kind is an agent-based simulation [1].

We have perused the existing surveys of agent-based simulation frameworks (ABS) with special respect to EUSAS-project goals. In the first round of the evaluation we reviewed a high number of various agent based platforms [1] based on published surveys and the information on the web. In the second round – “Evaluation by Implementation” - we evaluated in depth the two most promising ABS systems by implementing an exemplary scenario described in section 2, which reflects the main needs of the EUSAS-project.

Besides smooth incorporation in highly realistic virtual trainings, even more important was the ease of use in multi-parametric studies (Data Farming) where many instances of the same ABS run in parallel, each with different values of input parameters. The results of each run are stored in a repository for subsequent analysis.

Several ABS that we considered were based on Logo languages (derived from Lisp). Here, NetLogo [4] was the most relevant representative. Other platforms included Repast\(^1\) or Mason [9], which can run high number of agents by executing each agent in small steps. In contradistinction to step-based implementations, there are also event-based or thread-based modeling toolkits, such as CoJack\(^2\) or Jason\(^3\). Here, each agent is executed in a separate thread and behavior is updated based on events. The event-based approach is used in VBS2 serious game component, which we plan to use for virtual trainings in the EUSAS system. Step-based ABS platform, such as NetLogo, Repast or Mason, allow simulation of a higher number of agents, and models are easier to debug, although there is an extra effort involved in integrating them with the thread and event-based serious game component for the purpose of virtual training. Creation of a large number of threads (e.g. thousands) would be inefficient in any of the thread-based toolkits.

Since we did not have the resources to evaluate all the existing platforms by implementation, we first shortlisted the candidates based on the existing MAS surveys and then evaluated the two most promising candidates by implementing an exemplary human behavior scenario which represented our domain. Based on the surveys, MASON and NetLogo were identified as the two most promising systems, each with a slightly different strategy. Compared to MASON, NetLogo was more focused on educational purposes, but still with a good capability for simple and fast modeling, implementation, visualization as well as good visual analytical tools. Both MASON and NetLogo are step-based platforms using discrete-event simulation model.

Apart from simulations for multi-parametric studies, we also intend to conduct simulations where real humans can interact, in order to support virtual trainings. Therefore we have also explored the possibilities for integration with a virtual reality toolkit, such as VBS2.

---

1 http://repast.sourceforge.net/
3 http://jason.sourceforge.net/Jason/Jason.html
1.1 Existing Survey Literature

The most relevant survey of ABS is [7] from 2005, which tested 5 ABS on a simple (so called Stupid Agent) scenario [7]. The evaluated platforms were NetLogo, MASON, Repast, Swarm and Java Swarm. MASON was evaluated as the fastest. All the features could be implemented quite well but its extensions and support tools were not all in a good shape then. NetLogo was found to be the simplest for agent modeling and implementation with good analytical tools and visualization. According to our recent research, NetLogo and MASON have been the fastest evolving ABS platforms since then. Repast was evaluated quite high. Repast is a well known platform with current beta version of Repast Symphony, which would be worth to evaluate by implementation, however Repast has several implementations and it is not clear which version it would be best to evaluate. Repast claimed to support NetLogo models, so we tried to import our implementation of NetLogo model into Repast, but we did not succeed since errors cropped up during the import process. When Repast Symphony reaches a stable release, it might be a worthwhile candidate for evaluation.

In a 2002 study [5], Repast, Swarm, Ascape, Smalltalk, StarLogo and AgentSheet were compared. Only Repast can be considered from this list nowadays. The most recent survey of MAS platforms is [6] using similar approach to [7]. It covers many platforms we considered based on available literature. We do not provide the list here but they are listed in [2] and many of them are also listed on the Wikipedia page on agent-based simulation\(^6\). As already mentioned, some of these platforms were evaluated on a StupidModel Programming experience for execution speed as well as ability to fully or partially implement the chosen features. StupidModel\(^7\) was broken down into 16 small tasks. It was implemented also in EcoLab C++ based Platform [8] and showed that EcoLab\(^8\) was capable of handling this model with similar performance as MASON but with worse GUI capability. StupidModel, however, is not fully relevant for our purposes. We decided to evaluate MASON and NetLogo by implementing our exemplary scenario (section 2), a simplified generic version of the kind of scenarios envisaged for human modeling in the EUSAS-project.

1.2 Evaluated Features

In order to evaluate the chosen simulation frameworks, we have defined 12 generic evaluation aspects on which we focused while implementing the scenario. These points are generic and could be relevant for other kinds of simulations as well, but we have evaluated them specifically in the context of implementing a typical human behavior model:

- **Loading and Representing the Environment and the Scenario:** Here we describe the representation and implementation of the scenario and the physical environment. We also discuss the possibility to load the environment model from GIS data as well as support for 3D, 2D and layered environments.

---

\(^6\) http://en.wikipedia.org/wiki/Comparison_of_agent-based_modeling_software
\(^7\) http://condor.depaul.edu/slytinen/abm/StupidModel/
\(^8\) http://ecolab.sourceforge.net/
Creating and Representing Agents: We discuss how to create, represent and implement agents in the evaluated system, and how the agents perceive other agents or their environment.

Behavior Implementation: Here we focus on behavior representation and implementation in the evaluated systems.

Movement Implementation: In this point we discuss the support for the physical movement of the agents in the environment, how they pass around obstacles or how a coordinated movement of crowd is supported. This is related to Flocking\(^9\) or Steering\(^{10}\) behavior\(^{11}\) of agents.

Visualization: Support for the simulation visualization, but also for running the simulation with no visualization (especially for Data farming purposes).

Parameterization: In order to run parametric studies (Data Farming), we have evaluated ABS support for simulation parameterization.

Model check-pointing: Support for model check pointing – stopping, storing, loading and running simulation from the previously stored break-point.

Analytical Tools: Support and analytical tools of ABS are discussed here.

Logging: To analyze multi-parametric studies and the measures of effectiveness, we need to log the progress of the simulation. We discuss here the ABS support for logging.

Performance: We discuss the perceived performance of ABS. In addition we provide performance measures for NetLogo and MASON for 10, 100, 1000 and 10000 civilian agents.

Standards: We discuss possible related standards such as HLA or FIPA.

Development Environment of evaluated platforms is discussed as well.

2 Human Behavior Modeling: Exemplary Scenario

In order to support tool evaluation with reference to the needs of human behavior modeling and the EUSAS-project as described in the introduction, an exemplary scenario [3] had to be defined. Hence, the exemplary scenario had to feature relevant aspects of human behavior in a given context, deriving from real world observations, and thereby reflecting the basic properties of the application context set by the EUSAS-project, but also to be kept as simple as possible in order to keep the implementation effort low and to enable rapid prototyping. Additionally the scenario should provide sufficient space for scenario evolution and should contain reactive and deliberative control structures for involved agents. Since the main focus of the paper

\(^{9}\) [http://www.red3d.com/cwr/boids/](http://www.red3d.com/cwr/boids/)

\(^{10}\) [http://opensteer.sourceforge.net/](http://opensteer.sourceforge.net/)

\(^{11}\) [http://www.shiffman.net/teaching/nature/steering/](http://www.shiffman.net/teaching/nature/steering/)
lies upon technical evaluation of the simulation frameworks in order to select the one supporting the needs of the EUSAS-project best, the following description is intended to provide an overview about the scenario elements, not to present the underlying formal model for the different aspects of agent behavior.

The scenario comprises a civil protester and a soldier, both represented as agents acting in a common environment.

The environment is a 2D grid composed of quadratic cells sized 0.5m x 0.5m. Each cell is labeled to describe its nature, respectively the actions which may take place if an agent enters the cell. The labels are: fight area, stone picking area, safety area, soldiers area, barrier.

Depending on the internal state of the civil protester agent, he resists in a predefined safety area of the environment or shows aggressive actions against the local authority represented by the soldier agent. Aggressiveness of the civilian protester is expressed by picking up a stone, approaching the soldier agent and throwing the stone towards him. Fearful behavior in contrast is expressed by flight reactions into a predefined safety area. The soldier agent's behavior is based on a text book case, hence he behaves according to a given rule set and is not triggered by any human motives. Being threatened, the soldier agent is allowed to take countermeasures against the threatening civilian agent.

The behavior of the civilian agent requires the following elements: stimulating events in the environment, motives, action plans and predefined behavior patterns. Based on the psychological considerations in [11], the civilian agent architecture contains three motives: fear, anger and an observe-motive. The theory of cognitive appraisal for emotions [10] serves as a theoretical basis for modeling the emergence and temporary course of the emotional motives anger and fear. Accordingly, stimulating events in the environment (e.g. movements or actions of the soldier agent) being perceived and cognitively evaluated by the civilian agent influence the intensity of his emotional motives fear and anger. The concrete computation of the corresponding motive intensities is done with the help of differential equations. The observe-motive can be regarded as "fall-back-motive" with constant intensity. All available motives compete against each other; the motive with the highest intensity dominates the other motives and determines the concrete shape of behavior that the civil agent shows at a certain point of time.

Both the civilian and the soldier execute their actions according to individual internal actions plans. An action is defined as a non-interruptible, time-consuming operation performed by an agent. For each action, a set of preconditions is defined. An action plan is a list of actions to be performed one after another. Action plans can get interrupted. This happens if the dominant motive changes or the precondition for the next action in the plan is not fulfilled. In this case, the whole action plan gets rejected and the agent is forced to determine a new goal to reach and, consequently, to construct a new action plan.
3 Evaluation through Implementation

In this chapter we describe our experience with implementing the exemplary scenario described in section 2 in both MASON\(^\text{12}\) and NetLogo\(^\text{13}\). Scenario environment is grid based but in both NetLogo and MASON we implemented it as continuous, so agents interact and move continuously with a small defined discrete step. Both the evaluated systems are step-based simulation systems based on discrete-events. Although VBS2 (the serious game training component) is not directly competing with NetLogo or MASON, the chosen candidate would be later integrated with it for training purposes. Therefore, at appropriate places, we also refer to our implementation experiments with VBS2 and discuss potential integration issues. Figures below show screenshots of the exemplary scenario in NetLogo (Figure 1, left) and MASON (Figure 1, middle).

![Fig. 1. Left: Exemplary Scenario implemented in NetLogo with variable sliders and charts; Middle: Exemplary Scenario in MASON; right: MASON console window, where inspector of agent variables is open.](image)

3.1 Loading and Representing Environment

The NetLogo world is a two-dimensional grid of "patches". NetLogo supports three-dimensional environments, but the status of this feature is still experimental. Patches are the individual squares in the grid. Each patch is a square piece of "ground" over which the agents (turtles) can move. The way the world of patches is connected can change. World can be wrapped horizontally, vertically or in both directions (torus). In our exemplary scenario, we wanted to find a way how to load a map of areas into the NetLogo 2D world. We found it very convenient to represent the simulation scenario map by a bitmap image, where each pixel represents a patch of the world and the pixel color defines an area to which the patch belongs. To load the scenario map into NetLogo, we used a built-in command `import-pcolors-rgb <file>`, which reads an image file, scales it to the same dimensions as the patch grid (while maintaining the original aspect ratio of the image), and transfers the resulting pixel colors to the patches. After we load the map into the NetLogo world, we were able to refer to the patches from a desired area by the patch/area color.

\(^{13}\) [http://ccl.northwestern.edu/netlogo/](http://ccl.northwestern.edu/netlogo/)
In MASON we had to create a text file with an environmental matrix, i.e. with numbers representing the areas of the scenario environment. We had to implement the loading of this environment into MASON’s environmental structures. Environment in MASON can be 2D or 3D, and for both a variety of demo implementations is available. We chose 2D environment and started with IntGrid2D, which can hold a matrix of integers. After the implementation we found that the agents were moving too jerkily (jumping abruptly from one field to another) so we changed the environment into 2 layers, where the agents were moving in Continuous2D layer while the area definitions remained in IntGrid2D. While creating the continuous layer, we were able to define a discretization of the area which helped us to integrate the two layers. So in MASON the users can define multiple layers of continuous or discrete environments to represent their scenario environment. These layers (environment variables) need to be defined in the main class representing the simulation, which, in turn, has to be derived from the SimState class. Through the instance of this class the agents can access the current state of the environment. We have created a Demo class which extends SimState and consists of people variable (Continuous2D layer) holding the agent positions and grid variable (IntGrid2D) defining the physical environment.

GIS support. In recent releases, NetLogo was equipped with a GIS extension\(^\text{14}\) for loading the vector GIS data (points, lines, and polygons) and raster GIS data (grids). We have tested it successfully on OpenStreetMap\(^\text{15}\) data.

MASON did not have a GIS support for a long time. This has changed in the past few months and currently MASON supports the GeoMason\(^\text{16}\) extension, which we intend to test in the near future.

Both NetLogo and MASON can satisfy the modeling needs regarding the physical environment. Now they both have a GIS support, which simplifies loading of the existing environments to these tools and integration with VBS2 training component.

### 3.2 Creating and Representing Agents

A world in NetLogo is made up of agents, where each agent can perform its own activity simultaneously with and independently of other agents. There are four types of agents in NetLogo: turtles, patches, links and the observer. Except the turtles, all the other agent types are static. We represented soldiers and civilians as turtle agents. We also represented stones as turtle agents, to easily simulate their throwing.

An agent in MASON is an instance of a Java class that implements Steppable interface, where the method `step(SimState state)` needs to be implemented, representing the agent behavior. This method represents one agent simulation step in the environment and is called by the scheduler. We have implemented 3 agent classes (types): Soldier, Civilian and Stone. Compared to NetLogo, in MASON we can implement each agent in a separate file/Java class, which provides for better organization of software code. Agent instances are created in the same way as any

\(^{14}\) http://ccl.northwestern.edu/netlogo/docs/gis.html

\(^{15}\) http://www.openstreetmap.org/

\(^{16}\) http://cs.gmu.edu/~eclab/projects/mason/extensions/geomason/
Java class instance, and are then scheduled by the SimState simulation. Once scheduled, we can retrieve their reference (pointer) which we need in order to destroy the agent, e.g. when a Civilian is arrested and should disappear, or when a stone is thrown and no longer needed. We create the Civilians and Soldiers inside the Demo class. A stone agent is created when the Civilian enters the stone picking area and is destroyed when it hits the Soldier or (if it misses) after a few more simulation steps. VBS2 agents can be created through script commands, ASI, VBS2Fusion, or through special tools like OME (Offline Mission Editor) and RTE (Real Time Editor).

3.3 Behavior Implementation

In NetLogo, an agent consists of a function describing its behavior and a number of attributes (agent variables), which describe the agent state. The agent behavior can be implemented in several ways. NetLogo code examples include a state machine implementation approach using a turtle variable and the RUN command. A state machine consists of a collection of states with a different action associated with each state. In our implementation of the scenario, we used a different approach. We have used turtles to represent the soldier and civilian agents and we also defined some specific variables for these kinds of agents. The behavior of our agents depends on the agent variables, which hold the state and motive variables defined in scenario. In each simulation step, we recalculate all the agent motive variables reflecting the actual state in the environment and choose the motive with the highest value as action leading. The action related to the action leading motive is then executed.

In MASON, the agent behavior is implemented and called via step(SimState state) method. The parameter SimState represents the simulation instance, holding also the defined properties of the environment and simulation.

The simplest behavior is that of the Stone agent. Stone agent is created when the Civilian enters the stone picking area. Then it is just carried by the Civilian agent along its path. Civilian and Soldier are another agent types implemented according to scenario from section 2.

Agent behavior in MASON is implemented through the step() method, which is invoked at each simulation step for the environment as well as for the agents and their internal components (fear, anger, etc.). The agent can access the environmental state via the SimState instance passed to the step() method. The agent can also invoke the getObjectsWithinDistance method on Int2D or Continuous2D environment properties to locate the appropriate objects depending on its intentions.

VBS2 agents are represented as Finite State Automata or Finite State Machines. Agents behavior can be implemented using an FSM editor, by scripting in a text editor, through Application Scripting Interface or, finally, by VBS2Fusion API.

Overall, we felt that both NetLogo and MASON had the needed support for the behavior modeling. In both cases, the behavior implementation had to be step-based, which differed from VBS2 and other virtual reality tools that were thread and event-based. This difference may have an impact on the integration and behavior implementation.
3.4 Movement Implementation

NetLogo offers a lot of built-in variables and commands, which make the implementation of the agent movement easy and straightforward. One can define location by `setxy <x> <y>` (e.g., its initial position in the environment), by `set heading towards <agent>` to set the heading of a civilian to nearest stone for example or by `forward <distance>` to move agent forward in the heading direction by specified distance. Another useful command that we used a lot is `distance <agent>`.

To the best of our knowledge, the movement algorithms are not supported well in MASON. All we could do in MASON was to set up a new location for the agent in each step. In NetLogo, movement is supported much better because of its turtle nature. So in MASON we had to implement the basic step-wise movement towards the target. The implementation of Flocking or Steering behavior (movement) is also not directly supported. However, Flocking is implemented in one of the MASON demos called Flockers. We will try to reuse it and test it. For flocking behavior in NetLogo, the programmer simply defines the closest distance among the agents and NetLogo steers the agents so that this distance is guaranteed.

Agent movement in VBS2 is planned via the A-star algorithm. VBS2 is able to plan the optimal path also using the waypoints.

Overall, NetLogo definitely has a better support for agent movement (at least heading towards is supported) than MASON. In MASON, a few sample implementations are available but not directly supported. In addition NetLogo offers built-in turtle commands for hill climbing and descending into valleys according to a variable value of patches around the turtle. There is also a support for "cone of vision" in NetLogo, which allows a turtle to set its viewport (vision angle and distance) and ask for agents that fall in the cone.

3.5 Visualization

In NetLogo, vector shapes are used to visualize turtles. Vector shapes are built from basic geometric shapes (squares, circles, and lines) rather than from a grid of pixels. Vector shapes are fully scalable and rotatable. NetLogo caches bitmap images of vector shapes (magnified by a factor of 1, 1.5, and 2) so as to speed up execution.

NetLogo can be invoked and controlled by another program running on the Java Virtual Machine. It is possible to embed NetLogo models in a larger application. There is an API for this purpose, but it is considered as experimental and is likely going to change in the future releases of NetLogo. When running NetLogo models by API, it is possible to turn off the GUI.

In MASON, a very useful feature is the strict separation of visualization and simulation. In order to run the simulation with the visualization one has to create a new class derived from the `GUIState` class, which then instantiates the `SimState` implementation. For visualization layers one can use Portrayals, which usually match the variables representing the environment. One can define how their values will be mapped to colors or how to draw the agents. We have implemented only 2D visualization, but 3D is also possible and included in MASON demos.
VBS2 is used to show highly realistic 3D environments. There is a problem with smoothly visualizing atomic actions in special cases, e.g., when a civilian wants to throw a stone but the leading motive changes, so it starts turning back towards the safety area in the middle of a throwing action.

Overall, both MASON and NetLogo have equally good support for visualization, but MASON supports 3D for a longer time. In MASON, multiple displays can be used and models can be run fully independently of visualization. In both NetLogo and MASON one can switch off the visualization. But only in MASON the simulation models are truly independent from the visualization, which makes it much faster – an important factor for multi-parametric studies (data farming).

3.6 Parameterization

NetLogo offers a tool called BehaviorSpace, which can run one model many times, systematically varying the model’s settings and recording the results of each model run. BehaviorSpace lets the user to explore the model’s "space" of possible behaviors and determine which combinations of settings cause the behaviors of interest. User can parameterize a particular variable by specifying a list of all its possible values, by defining an initial value, final value and increment, or the variable can be randomly varied within a specified range.

Since MASON is built in Java, parameterization of simulation can be easily implemented. Direct support for parameterization of simulation is provided in the form of a tutorial\(^{17}\).

Both systems support the parameterization needed for our multi-parametric studies (data farming). With MASON it is probably easier to achieve a massive run-time job-level parallelism. On top of that, MASON also performs well when running more instances on a single machine with more CPU cores, and has a strong separation of the visualization and the behavior model.

3.7 Model check pointing

When running a model with NetLogo GUI, it is possible to manually stop the simulation and save (export) its whole world state into a file. NetLogo automatically saves all the values of all the variables, both built-in and user-defined, including all the observer, turtle, and patch variables, the drawing, the contents of the output area (if it exists), the contents of any plots and the state of the random number generator. The resulting file can be then read back into NetLogo and simulation can continue from the saved state. This export/import functionality is provided by the built-in commands
\[
\text{export-world <file> and import-world <file>}.\]

MASON too has a good support for the model check-pointing – storing simulation at any time to a disk file. Later the model can be re-loaded and the simulation re-started from the same point. We have tested this feature and it worked well.

\(^{17}\) http://www.cs.gmu.edu/~eclab/projects/mason/extensions/webtutorial1/
VBS2 game can be saved at any time and there is no problem in restarting it from several checkpoints made during the game to test alternative branches of the scenario.

Both NetLogo and MASON support the model check-pointing, but MASON also claims cross-platform compatibility.

3.8 Analytical Tools

Results of the NetLogo simulation can be displayed to the user in the form of a plot or a monitor. The first is the traditional way of displaying data in two or three-dimensional space. Monitor is another popular form consisting of a number of frames, each of which represents a concrete attribute of a simulation and its current numerical value. Users can export this data to a file in order to read and analyze it later with other applications, e.g. a spreadsheet. We have tried to visualize some state and motive variables of a civilian agent in plots (see charts on left side of Figure 1).

NetLogo Profiler extension helps measuring how many times the procedures in the model are called during a run, and how long each call takes. The profiler extension is new and experimental and is not yet well tested or user friendly. NetLogo System Dynamics Modeler is used to describe and understand how things in a model relate to one another. Instead of modeling behavior of individual agents and use them as the basic building block of a model, the populations of agents is described as a whole by differential equations.

MASON simulations can run directly as Java code without visualization. When running with visualization, simulations are controlled through the Mason Console (Figure 1, right) that allows starting, pausing and stopping. Users can load the stored models and run them from specific checkpoints. They can also record the simulation as a movie or take a screenshot. It is possible to set delays and choose one of multiple displays. Multiple displays are used when we need to have more than one view of the simulation. Similarly as in NetLogo, the users can inspect all the public agent variables (but setter and getter methods need to be implemented). Their changes can be displayed as a Chart (JFreeChart extension) or streamed into a file.

VBS2 comes with the AAR (After Action Review) tool, which can be used for replaying and analyzing the whole mission to find crucial moments in the scenario. Here, NetLogo was a traditional winner, but now MASON also has a good support for the analysis of variables evolving in time by streaming or drawing charts.

3.9 Logging

NetLogo uses the Log4j package for logging. NetLogo defines eight loggers (Globals, Greens, Code, Widgets, Buttons, Speed sliders, Turtles, Links), which are configured through a configuration file.

To the best of our knowledge, MASON does not support the logging functionality directly. We have implemented it using log4j. In each agent we have implemented the logging method, which receives the text label (usually describing actions) as input and

18 http://www.cs.gmu.edu/~eclab/projects/mason/docs/tutorial0/index.html
outputs all the information about the agent – its location, variable states (fear, anger), motives and the text label. This provided us all the needed functionality for logging.

VBS2 has its own logging module, but there are also several script commands, which can be used for logging whatever else might be required.

NetLogo has a direct support for logging. In MASON one can use the existing Java libraries such as log4j to log the simulation data.

3.10 Performance

Performance of MASON was evaluated in [7, 8] and NetLogo in [7], where it turned out that MASON was the fastest platform. We have evaluated it by running our exemplary scenario with varying numbers of agents and extending the physical area so as to accommodate them properly. We achieved this by copying the same base scenario area 10, 100 or 1000 times by placing a new copy of the base area on top of each other. We have then tested the performance by running the simulation 10 times for 1000 steps. Since one base area accommodates 10 civilians and 5 soldiers, the evaluated numbers of agents were (1) 10 Civilians versus 5 Soldiers; (2) 100 Civilians versus 50 Soldiers; (3) 1,000 Civilians versus 500 Soldiers; and, finally, (4) 10,000 Civilians versus 5,000 Soldiers. In the last case we have run only 10 steps of the simulation for MASON. This step was not successful at all for NetLogo, because even with 1GB of Java heap space, NetLogo did not succeed in starting with 15,000 agents. Since NetLogo was much slower, we only run 10 steps for 1,500 agents.

In this way the systems were evaluated for up to 15,000 agents. This number did not include the stones, which were created and destroyed on demand. We have run the evaluation on the machine with two Intel(r) Core(TM) i7 CPU 860 2.80 GHz processors and 3GB RAM. The operating system was Windows 7 (32-bit version).

<table>
<thead>
<tr>
<th>Number of Agents</th>
<th>15</th>
<th>150</th>
<th>1500</th>
<th>15000</th>
</tr>
</thead>
<tbody>
<tr>
<td>NetLogo 1 step (ms)</td>
<td>0.48</td>
<td>27.60</td>
<td>18281.95</td>
<td></td>
</tr>
<tr>
<td>MASON 1 step (ms)</td>
<td>0.10</td>
<td>0.59</td>
<td>21.51</td>
<td>2474.30</td>
</tr>
</tbody>
</table>

Table 1: Performance evaluation summary

MASON and NetLogo performance is shown in Table 1. One simulation step took about 22 milliseconds for MASON and about 18 seconds for NetLogo for the middle option (No.3) with 1,500 agents. So MASON was almost 850 times faster. MASON speed is quite impressive and acceptable for real-time operation with virtual reality tools for about a thousand agents. NetLogo could be used well for a hundred of agents. While evaluating the performance we have switched off the logging for both MASON and NetLogo. With logging to file, the performance of MASON was 2-3 times slower. With logging both to file and to console the execution was 9-10 times slower. During the actual simulation the logging is needed, but the execution time of one step with 1,500 agents is still under 1/10 of second (about 66 milliseconds), which is still acceptable. For 15,000 agents, one simulation step took about 2.5 seconds for MASON (for NetLogo it did not even start), which is not acceptable for virtual reality trainings, but still acceptable for (off-line) Data Farming. All the
simulations were executed without GUI, but even with GUI the time of the simulation was still acceptable for 150 agents for both NetLogo and MASON. We did not measure and evaluate the exact time requirements of the simulations with GUI. In general, MASON is much faster than NetLogo. Additionally, we have tested the MASON performance on a single machine with four MASON instances running in parallel. Intel Core i7-720QM (4 cores) and 8GB RAM machine was used. One run of a single instance of MASON was 3.74 times faster than this parallel execution of four instances, which is a very good result. We did not perform this test for NetLogo.

In our test of VBS2, we have used the FSM combined with scripting implementations and the conclusion was that VBS2 could run 100 civilians and 20 soldiers with no delays at all (just in the initialization of the scenario there were some delays). We did not test VBS2Fusion, which suppose to be 200 times faster than ASI.

3.11 Standards

In this section we discuss related standards such as HLA or FIPA and their support in the evaluated platforms.

FIPA standards\textsuperscript{19} are relevant mainly for mobile and intelligent autonomous agents and are not so much related to agent based simulation. FIPA covers agent communication, management and transportation (for mobile agents). For agent based simulation only agent communication can be relevant, but in simulations this is limited to a few concrete communication messages so it is not so crucial whether an ABS supports FIPA or not. Neither NetLogo nor MASON support FIPA standards.

DIS and HLA standards\textsuperscript{22} are more relevant for ABS, especially if we want to integrate realistic civilian simulation with soldier/police virtual training as intended in EUSAS project. VBS2 serious game supports both HLA and DIS. Anyhow, rather than HLA or DIS, we plan to use the plug-in functionality in VBS2 and CORBA\textsuperscript{23} technology for real-time communication between ABS and VBS2 in EUSAS project, which would be easier to develop (e.g. no need to create a FOM - Federation Object Model). However since MASON is Java based, HLA based integration can be supported by using poRTico\textsuperscript{24} or Java port of CERTI\textsuperscript{25} for example. NetLogo, integration through HLA would be also possible but not so straightforward.

3.12 Development Environment

In multi-agent systems developers face problems with debugging the agents since they run in separate threads. Both NetLogo and MASON\textsuperscript{26} are step based, so models can be easily debugged as any procedural or object oriented program.

\textsuperscript{19} http://fipa.org/specifications/
\textsuperscript{22} http://www.sisostds.org/ProductsPublications/Standards/IEEEStandards.aspx
\textsuperscript{23} http://www.corba.org/
\textsuperscript{24} http://www.porticoproject.org/
\textsuperscript{25} https://savannah.nongnu.org/projects/certi/
\textsuperscript{26} In MASON, agent routine (step) is scheduled as an event, but there is only one event scheduled at one time.
NetLogo has its own development environment, which offers a lot of usable tools such as the source editor, interface builder or agent monitors. NetLogo environment allows users to run models and inspect their properties. Debugging can be done mainly by executing one step of simulation and watching how the agent variables change and how the visualization of the simulation changes. Developer can interact with the model by Command center on-the-fly, where it is possible to execute custom commands.

MASON is Java based library. Any Java IDE can be used to develop in MASON. We have used Eclipse\(^27\). There is also tutorial available on how to use MASON with Eclipse. Standard Java debugging procedures can be used easily to develop, debug and test MASON models.

Our experience is that simple well organized libraries such as MASON \(^9\) are easier for programmers familiar with Java than more complex ABS IDEs, such as Repast Symphony \(^1\).

### 4 Discussion and Conclusion

In this paper we have summarized literature surveys of ABS and evaluated two candidates – MASON and NetLogo by implementing exemplary human behavior scenario. Recently, there have emerged interesting new candidates, such as Repast Symphony or Janus\(^28\) with its JaSIM\(^29\) extension, which we might evaluate along these lines in the future.

Table 2 provides a summary of the evaluated features in MASON and NetLogo. Both are almost equal in many features. NetLogo is better in the physical movement support and some analytical tools. MASON is much faster, supports strong separation of visualization and behavior models, has a better support for 3D environment and is based on Java, which makes it far easier to integrate with other systems.

<table>
<thead>
<tr>
<th>Features</th>
<th>NetLogo</th>
<th>MASON</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>Logo, Java for simulation control</td>
<td>Java</td>
</tr>
<tr>
<td>Environment</td>
<td>2D, 3D experimental</td>
<td>2D, 3D</td>
</tr>
<tr>
<td>GIS support</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Movement</td>
<td>Heading angle + step</td>
<td>just set(x,y)</td>
</tr>
<tr>
<td>Stearing/Flocking Behaviour</td>
<td>Not directly</td>
<td>Not directly</td>
</tr>
<tr>
<td>Visualization</td>
<td>2D, 2D as 3D</td>
<td>2D, 3D</td>
</tr>
<tr>
<td>Run with no visualization</td>
<td>possible but not strictly separated</td>
<td>separated behaviour and visualization models</td>
</tr>
<tr>
<td>Parametrisation</td>
<td>possible</td>
<td>possible</td>
</tr>
<tr>
<td>Model check-pointing</td>
<td>Yes</td>
<td>Yes, platform independent</td>
</tr>
<tr>
<td>Analytical Tools</td>
<td>Charts, Streaming, variable bars, snapshot</td>
<td>Charts, Streaming, snapshot, video recording</td>
</tr>
<tr>
<td>Logging</td>
<td>support using log4j</td>
<td>not direct support but log4j can be used</td>
</tr>
<tr>
<td>Performance</td>
<td>good for tens of agents</td>
<td>good for thousands of agents</td>
</tr>
</tbody>
</table>

Table 2: Evaluated features summary

NetLogo has proved its reputation as an ABS platform where the simulation models can be implemented quickly and straightforwardly. A bit problematic is the development of complex models, which cannot be structured well – each source file is limited to include only one external source file. The integration with the serious game

\(^{27}\) [http://www.eclipse.org/](http://www.eclipse.org/)

\(^{28}\) [http://www.janus-project.org/](http://www.janus-project.org/)

\(^{29}\) [http://www.multiagent.fr/Jasim_Platform](http://www.multiagent.fr/Jasim_Platform)
component is difficult, because it would require developing a custom plug-in for NetLogo.

Regarding MASON, we have appreciated its rapid improvements over the past few years, with new plug-ins and tools (such as GIS support) continually being created. Its performance is impressive – it can support thousands of agents in one simulation. It is Java-based, which helps in its integration with the external systems (e.g. serious game component – VBS2). Similarly, the logging functionality can be implemented through other Java-based components, such as log4j.

Overall, we were greatly impressed by the NetLogo modeling support, functionality and the overall system, which makes it an extremely valuable tool for educational purposes, and for scientific model development and analysis. Had we simply looked for a handy standalone agent-based simulation tool for a limited number of agents, NetLogo easily could have been our choice. Regarding the specific goals and requirements of the EUSAS project, however, we had to conclude that MASON’s speed, flexibility and extensibility were more important and made it the best-suited candidate for the job.

Acknowledgments. The paper was supported by EDA project A-0938-RT-GC EUSAS (European Urban Simulation for Asymmetric Scenarios).

References

An Agent Infrastructure for Privacy-enhancing Agent-based E-commerce Applications

Jose M. Such, Agustin Espinosa, and Ana Garcia-Fornes

Departament de Sistemes informàtics i Computació,
Universitat Politècnica de València,
Camí de Vera s/n, 46022 València (Spain).
{jsuch,aespinos,agarcia}@dsic.upv.es

Abstract. Privacy is of crucial importance in order for agent-based e-commerce applications to be of broad use. Privacy can be enhanced minimizing data identifiability, i.e., the degree by which personal information can be directly attributed to a particular individual. However, minimizing data identifiability may directly impact other crucial issues in agent-based e-commerce, such as accountability, trust, and reputation. In this paper, we present an agent infrastructure for agent-based e-commerce applications. This agent infrastructure enhances privacy without compromising accountability, trust, and reputation.

1 Introduction

Agent-based electronic commerce refers to electronic commerce in which agent technologies are applied to provide personalized, continuously running, semi-autonomous behavior [11]. In agent-based e-commerce, agents encapsulate personal information describing their principals. They usually have a detailed profile of their principal’s names, preferences, roles in organizations and institutions, location, transactions performed, and other personal information. Moreover, agents carry out interactions on behalf of their principals, so they exchange this personal information.

Privacy is of great concern in the era of global connectivity (everything is inter-connected anytime and anywhere) with almost 2 billion world-wide users with connection to the Internet as of 2010\(^1\). Recent studies show that 92\% of users are concerned or very concerned about privacy [37]. Moreover, almost 95\% of web users admitted that they have declined to provide personal information to web sites at one time or another when asked [17]. To our knowledge, privacy is seldom considered in the Multi-agent Systems research field and, in particular, in agent-based e-commerce applications. This leads to applications that invade individuals’ privacy, causing concerns about their use.

Two information-related activities can represent a major threat for privacy: information collection and information processing [26]. Information collection refers to the process of gathering and storing data about an individual. For

\(^1\) http://www.internetworldstats.com/stats.htm
instance, an attacker can be listening to the messages that two agents exchange over the network and simply gather the information that is in the content of these messages. Applications need to be secure to avoid undesired information collection \[16\].

Information processing refers to the use or transformation of data that has already been collected \[30\], even though this information has been collected by mutual consent between two parties. For instance, a vendor could have a complete profile of a customer containing relevant data collected from the purchases made by the customer’s agent. The vendor can then use information filtering techniques to obtain detailed information on the customer’s tastes. Then, the vendor can infer which goods the customer is more willing to acquire and offer them in advance through personalized advertising. Moreover, the vendor could even incur in price discrimination practices, i.e., the vendor could charge different prices to different customers depending on the desire that this customer has to acquire a product according to their tastes. Information processing can be avoided by means of minimizing data identifiability, i.e., minimizing the degree by which personal information can be directly attributed to a particular individual \[30\].

Minimizing data identifiability may have a direct impact on accountability. Accountability refers to the ability to hold entities responsible for their actions \[4\]. Accountability usually requires an unambiguous identification of the principal involved. Thus, this principal can be liable for her/his acts. Commercial systems emphasize accountability because, in these environments, principals can be subject to serious losses such as money loss. Moreover, the sense of impunity generated by the lack of accountability could even encourage abuse. Thus, accountability is of crucial importance for agent-based e-commerce because it helps to promote trust in agent-based e-commerce applications, which is needed for principals to be willing to engage with and delegate tasks to agents \[11\].

There is also the need to equip agents with models to reason about and assess trust towards other agents in an agent-based e-commerce application \[12\]. These models allow agents to select the best and most reliable partnership in a specific situation and to avoid partners of previous unsuccessful transactions. However, minimizing data identifiability may also have a direct impact on trust and reputation models. The ability to hold multiple pseudonyms (as is sometimes required to minimize data identifiability) causes the well-known identity-related vulnerabilities of most current trust and reputation models \[6\]. These vulnerabilities can place the system in jeopardy, causing significant money loss.

In this paper, we describe the support that the Magentix2\(^2\) Agent Platform (AP) provides for enhancing privacy in agent-based e-commerce applications. This support enhances privacy while preserving accountability and avoiding identity-related vulnerabilities of trust and reputation models. The remainder of this paper is organized as follows. Section 2 presents related relevant works. Section 3 gives a brief overview of the Magentix2 AP. Section 4 presents the Magentix2 AP.

\(^2\) http://users.dsic.upv.es/grupos/ia/sma/tools/magentix2/index.php
gentix2 agent identity management support. Section 6 presents an application scenario. Finally, Section 7 presents some concluding remarks and future work.

2 Related Work

2.1 Privacy-enhancing Agent Platforms

In order to avoid undesired information collection, sensitive personal information must be protected from access by any other third party that is different from the agent to which the information is directed to. Confidentiality is a security property of a system that ensures the prevention of unauthorized reading of information [31]. In distributed environments, confidentiality usually means that sensitive information is encrypted into a piece of data so that only parties that can decrypt that piece of data can access the sensitive information.

There are many Agent Platforms (APs) developed by the agent community – for an overview of current APs and the features they provide refer to [2]. However, only a few of them currently take security concerns into account. For instance, Jade [18], Magentix3 [32], AgentScape [24], SECMAP [34], Tryllian ADK [38], Cougaar [22], SeMoA [28], and Voyager [27] are security-concerned APs.

Current security-concerned APs provide confidentiality for the messages exchanged by the agents running on top of them. To this aim, APs use existing secure data transfer technologies such as Kerberos [21], SSL [15], and TLS [10]. These technologies allow the encryption of messages before transferring them and the decryption of messages once they are received. As a result, if an agent A sends a message to an agent B using these technologies, A is sure that B will be the only one able to read this message.

Confidentiality is a necessary condition to preserve privacy, but it is not sufficient. It prevents undesired information collection from unauthorized third parties. If an agent A sends personal information to an agent B in a confidential fashion, external third parties will not be able to access it. However, agent B will obviously receive this personal information. The point is that agent B can then process the received personal information, unless specific measures for preventing information processing are adopted before sending this information.

Most of the work for protecting against the processing of information already collected is based on minimizing data identifiability. Identifiability can be defined as “the degree to which (personal) data can be directly linked to an individual” [30]. The degree of privacy of a system is inversely related to the degree of user data identifiability. The more identifiable data that exists about a person, the less she/he is able to control access to information about herself/himself, and the greater the privacy risks. Identifiability ranges from complete identification to anonymity.

Note that the support we present in this paper is for Magentix2, which is a completely redesigned version of Magentix [32].
Pseudonymity [8] is the use of pseudonyms as identifiers. A pseudonym is an identifier of a subject other than one of the subject’s real names [23]. The most important trait of pseudonymity is that it comprises all degrees of identifiability of a subject (from identified to anonymous) depending on the nature of the pseudonyms being used. Complete identification is when the linking between a pseudonym and its holder is publicly known. Anonymity can be achieved by using a different pseudonym for each different transaction. This is known as transaction pseudonyms [8]. For instance, two agents A and B act as a customer and a vendor, respectively, in a marketplace. Agent A can use a different pseudonym (e.g., a random generated numeric identifier) for each specific transaction with agent B. Hence, Agent B collects information about the transactions performed but is unable to relate different transactions to each other or relate any of these transactions to agent A. However, e-commerce transactions themselves can include information that can be used to relate different transactions to each other and to agent A, e.g., the credit card number to perform the payments and the shipment address may be the same for different transactions. We assume that anonymous payments [7] and privacy-preserving delivery systems [1] are used to avoid this.

Only a few of the security-concerned APs explained above implement some kind of support for pseudonymity. Magentix, Secmap, AgentScape, and Cougaar allow agents to authenticate each other using their unique agent identity. With this identity, agents can act pseudonymously, i.e., agents can act on behalf of their principal without using the identity of their principal. However, agents cannot hold more than one pseudonym, i.e., principals should use a different agent each time they want to use a different pseudonym.

Warnier and Brazier [36] also present a mechanism for the AgentScape AP that offers pseudonymity by means of what they call handles. Handles are pseudonyms that agents can use to send/receive messages to/from other agents. At will, agents can request new handles to the AP. Moreover, the AP is the only one that knows the association between handles and GUIDs (global unique identities of the agents). An agent can also obtain anonymity by simply using a different handle for each transaction (transaction pseudonyms). AgentScape also offers an automatic anonymity service. Agents can send messages anonymously without having to manage pseudonyms. This service is provided by agents called anonymizers. When an agent wants to send a message anonymously, this message is redirected to an anonymizer. Then, this anonymizer is in charge of removing the original handle of the sender from the message, replacing it with another (possibly new) handle, and sending the message to the intended recipient. If the intended recipient replies, this reply is forwarded to the sender of the original message. The original sender of the message must notify when a transaction ends. For each new transaction the anonymizer generates a new handle.

In order to avoid a lack of accountability that could cause a sense of impunity and encourage abuse, AgentScape and Magentix keep track of the association between principals and pseudonyms. The main drawback of this approach is that the AP itself (including the anonymizer agents for the case of AgentScape)
must be trusted. This is because the AP knows the relation of pseudonyms to each other and to the principal involved. Although this is needed for ensuring accountability (agent principals can remain liable for their agents' behaviour even when pseudonyms are used), this usually implies that the organization or company that hosts the specific marketplace (e.g. eBay) knows the association of pseudonyms to each other and to principals. Therefore, this organization or company can collect and process information about the principals that run their agents on the marketplace.

In this paper, we present the support for pseudonymity provided by Magentix2. Magentix2 allows agents to use as many pseudonyms as they need to preserve their privacy by avoiding information processing. We refer to these pseudonyms as regular pseudonyms. Moreover, Magentix2 does not keep track of the association between principals and pseudonyms. It relies on trusted external identity providers to keep this information.

2.2 Trust and Reputation

Trust and reputation play a crucial role in agent-based e-commerce applications. There have been many proposals for trust and reputation models [25, 29]. These models are usually based on the assumption that identities are long-lived, so ratings about a particular entity from the past are related to the same entity in the future. However, when pseudonymity techniques are used, this assumption is no longer valid. For instance, an agent that has a low reputation due to its cheating behavior may be really interested in changing its pseudonym and restarting its reputation from scratch. This is what Jøsang et al. [19] called the change of identities problem. This problem has also been identified by other researchers under different names (e.g. whitewashing [6]).

Kerr and Cohen [20] also point out the fact that entities could create new pseudonyms at will, not only after abandoning their previous identity but also holding multiple identities at once. This is known as the sybil attack [19]. An example of this attack could be an agent that holds multiple pseudonyms in a marketplace and attempts to sell the same product through each of them, increasing the probability of being chosen by a potential buyer.

These vulnerabilities can cause principals to lose money. A possible solution for these vulnerabilities is the use of once-in-a-lifetime pseudonyms [14]. Agents can only hold one once-in-a-lifetime pseudonym in each marketplace. Therefore, they cannot get rid of the trust and reputation ratings they got from other agents in the marketplace. A model for agent identity management based on once-in-a-lifetime pseudonyms has been proposed in [33]. Magentix2 implements this model (as detailed in section 4). Agents in Magentix2 can have two kinds of pseudonyms: permanent pseudonyms (once-in-a-lifetime pseudonyms), which avoid identity-related vulnerabilities; and regular pseudonyms, which agents can use without any limitation in number to obtain their desired degree of privacy.
Magentix2 Agent Communication

The Magentix2 AP focuses on providing support for open MAS. Magentix2 uses AMQP [35] as a foundation for agent communication. This standard facilitates the interoperability between heterogeneous entities. Magentix2 allows heterogeneous agents to interact with each other via messages that are represented following the FIPA-ACL [13] standard, which are exchanged using the AMQP standard.

Magentix2 uses the Apache Qpid open-source implementation of AMQP for Agent Communication. Apache Qpid provides two AMQP servers, implemented in C++ (the one we use) and Java. Qpid also provides AMQP Client APIs that support the following languages: C++, Java, C# .NET, Ruby, and Python. Qpid allows distributed applications made up of different parts written in any of these languages to communicate with each other. What is more, any client that is developed using one of the Qpid Client APIs is able to communicate with any client that is developed using any other AMQP-compliant API via any AMQP server implementation, as long as both server and clients implement the same version of the AMQP standard.

Figure 1 shows an overview of the Magentix2 agent communication architecture. Magentix2 is composed by one or more (in this case federated) AMQP Servers (QPid brokers). Magentix2 agents act as AMQP Clients (using Qpid Client APIs) that connect to the Qpid broker and are then able to communicate with each other. Magentix2 agents can be located in any Internet location, they only need to know the host on which the Qpid broker (or one of the federated Qpid brokers) is running.

Fig. 1. Magentix2 Agent Communication Architecture

---

4 http://users.dsic.upv.es/grupos/ia/sma/tools/magentix2/index.php
5 http://www.amqp.org/
6 http://qpid.apache.org/
Magentix2 provides a Java library, which is called the Magentix2 Agent Library (MAL), to facilitate the development of agents. This API allows agent programmers to specifically focus on creating FIPA-ACL messages and sending and receiving them, without dealing directly with the Qpid Client Java API. Currently, this API is only written in Java, but the existence of multiple QPid Client APIs for several programming languages enables the development of agents written in different programming languages. What is more, any proprietary implementation that follows both AMQP and FIPA-ACL standards would be interoperable with Magentix2 agents.

4 Magentix2 Agent Identity Management

Magentix2 implements the agent identity management model presented in [33]. This model is based on the concept of partial identity. A partial identity can be seen as a set of attributes that identifies an entity in a given context. They are composed of a pseudonym that is unique within a context and other attributes that describe the entity within that context (roles, location, preferences, etc.).

This model considers two kinds of partial identities: permanent partial identities (PPIs) and regular partial identities (RPIs). A PPI must contain a permanent pseudonym (*once-in-a-lifetime* pseudonym) for a given marketplace. Thus, agents can only hold one PPI in this given marketplace. A RPI can contain a regular pseudonym that does not pose any limitation on the number of these pseudonyms per agent and per marketplace. Although both kinds of partial identities enable trust and reputation relationships, only PPIs guarantee that identity-related vulnerabilities are avoided. Therefore, agents will choose to establish trust and reputation through PPIs if they want to avoid identity-related vulnerabilities. If they want to avoid information processing, they can use as many RPIs as needed. For instance, an agent can use a different RPI for each different transaction (transaction pseudonyms).

This model also considers the concept of real identities. Real identities identify entities that can be liable for their acts in front of the law, such as human beings, companies, etc. Real identities are used for accountability concerns such as law enforcement. For this reason, real identities are restricted to only legal persons. A real identity, for example, would be: *Bob Andrew Miller, born in Los Angeles, CA, USA on July 7, 1975.* Software entities (intelligent agents, virtual organizations, etc.) cannot have real identities because, up to now, they cannot be liable for their acts in front of the law.

Magentix2 complies with the client part of the Identity Metasystem Interoperability standard. This standard specifies the interfaces for the secure web services provided by User-Centric Privacy-Enhancing Identity Management Sys-

\footnote{This may change in the future if they finally achieve some kind of legal personality, as suggested by [3]. In this case, they may have a real identity for accountability concerns as well.}

\footnote{http://docs.oasis-open.org/imi/identity/v1.0/identity.html}
tems [9]. These systems support the process of management of partial identities. They provide the following facilities:

- **Identity Providers (IdPs)**, which issue partial identities and validate these identities to other Relying Parties.
- **Relying Parties**, which are a set of APIs for verifying partial identities against an Identity Provider.
- **Identity Selectors**, which provide a simple way to manage partial identities and choose which partial identity to use in a given context.
- **Attribute Services**, which allow the specification of access control rights of relying parties over the attributes in a partial identity.

![Diagram of Magentix2 agent identity management support](image)

**Fig. 2.** The Magentix2 agent identity management support.

Figure 2 shows an overview of the Magentix2 agent identity management support. The Magentix2 Management Service (MMS) is a secure web service that acts as a Relying Party, i.e., it is able to request IdPs to verify partial identities. The MMS is in charge of dynamically signing digital certificates for agents to communicate securely in Magentix2 (as described in section 5). Agents request the signing of digital certificates to the MMS using one of their partial identities. The MMS must verify the partial identity that the agent used before signing the digital certificate.
The Magentix2 Agent Library (MAL) implements clients for Identity Selectors, Relying Parties, and Attribute Services. Therefore, agents in Magentix2 can select the partial identity to use in a given transaction, verify the partial identities of other agents, and specify access control for attributes in their partial identities.

![Diagram of Partial Identities of an agent](image)

**Fig. 3.** An example of the Partial Identities of an agent.

IdPs are classified according to the type of partial identities they issue. The Permanent Identity Provider (PIdP) is an IdP (or a federation of IdPs\(^9\)) that issues PPIs to the agents taking part in the specific marketplace. Agents must register using a real identity that the PIdP will not reveal to other agents or to Magentix2. The PIdP is also in charge of forcing agents to only hold a single PPI in this specific marketplace.

\(^9\) User-Centric Identity Management Systems support the federation of IdPs that belong to the same and also different remote security domains across the Internet. Therefore, a PIdP can be implemented as a federation of IdPs instead of only one IdP, minimizing the typical drawbacks of a centralized trusted third party, such as being a single point of failure (SPOF) and a possible efficiency bottleneck. Examples of identity federation standards are the Liberty Alliance Identity Federation Framework [http://projectliberty.org/resource_center/specifications/liberty_alliance_id_ff_1_2_specifications/](http://projectliberty.org/resource_center/specifications/liberty_alliance_id_ff_1_2_specifications/) and WS-Federation [http://www.ibm.com/developerworks/library/specification/ws-fed/](http://www.ibm.com/developerworks/library/specification/ws-fed/).
Regular Identity Providers (RIdPs) issue RPIs to agents. Agents request RPIs by providing either a real identity, or a PPI that RIdPs will not reveal to others. There is no limitation in the number of RIdPs per marketplace or in the number of RPIs per agent and per marketplace.

Figure 3 shows an example of an agent and its partial identities. The agent’s principal has a real identity with an attribute name Adam John Wilkes. Using this real identity, the agent has obtained a PPI from the PIdP that includes two attributes: name and role. This entity has also obtained N RPIs from N different IdPs. Some of the RPIs are obtained by providing a PPI (such as RPI 1) and other RPIs are obtained using a real identity (such as RPI N).

5 Magentix2 Secure Agent Communication

Agent communication in Magentix2 is based on AMQP. The AMQP standard specifies secure communication by tunneling AMQP connections through SSL [15] (so-called amqps). Apache Qpid implements SSL support for AMQP. SSL authenticates communicating parties based on digital certificates. Thus, it needs a configured Public Key Infrastructure (PKI). The Magentix2 PKI is set during installation time. Firstly, the Magentix2 certificate authority (MCA) is created. Secondly, certificates for the Magentix2 Management Service (MMS) and the Qpid Broker are created using this certificate authority. Digital certificates for agents are created automatically by the MAL and dynamically signed by the MCA through the MMS at execution time (as described below).

The MMS is a front-end of the MCA. It is implemented as a secure web service. The MMS is in charge of dynamically signing digital certificates for agents, which can use these certificates to communicate securely. The MMS service needs two inputs: the agent pseudonym and a non-signed digital certificate. The first input is the pseudonym in the permanent or regular partial identity (issued by a permanent or regular IdP) that the agent uses to invoke the MMS. The second input is a non-signed certificate that contains the agent’s public key (this is the certificate that is to be signed). The agent key pair (private and public key) and this certificate are created by the MAL locally for each agent and for each new partial identity.

The MMS produces one output: the digital certificate signed by the MCA. The MMS produces this output after: (i) verifying that the pseudonym is the same as the one in the partial identity used to invoke the secure web service; (ii) verifying the partial identity against the IdP that issued it; (iii) and finally signing the certificate using the MCA. Agents can then use this signed certificate to communicate to other Magentix2 agents. Figure 4 shows an example of an agent with pseudonym A that obtains a certificate from the MMS. Thus, agent A can communicate securely with agent B.

The AMQP connection of every agent to the Qpid broker is tunneled through SSL. Hence, the communication between two Magentix2 agents is provided with confidentiality and integrity out of the box. To ensure the authenticity of the sender pseudonym in a FIPA-ACL message (recall that in Magentix2 FIPA-ACL
messages are encapsulated into AMQP messages), an agent must verify that the pseudonym of the sender in the AMQP sender message field is the same as the pseudonym of the sender in the FIPA-ACL sender message field upon receiving a new message. This is performed automatically by the Magentix2 agent library.

6 Application Scenario

We describe a Business-to-Consumer (B2C) electronic marketplace where seller agents retail medicines to buyer agents. Privacy can be of great concern in this scenario. A principal may need to acquire different medicines but does not want these medicines to be linked to her/him. For instance, there are medicines that are only prescribed for one specific illness, such as asthma. Therefore, buying these medicines automatically discloses the illness that the principal is suffering from. A principal may prefer to conceal his/her real identity when acquiring such medicines. This is because she/he is probably concerned about her/his illnesses being in the public domain and affecting other aspects of her/his life such as finding a job.

The principal can instruct her/his buying agent to obtain a partial identity that is different from her/his real identity before entering the marketplace. IdPs act as independent third parties that must be trusted by both Magentix2 and the agents. To obtain new partial identities (PPIs or RPIs), agents must provide a real identity, or a PPI to IdPs. IdPs do not make the original partial identities
available. Therefore, the rest of the agents in the marketplace and Magentix2 itself are, a priori\(^\text{10}\), not able to link a partial identity to the corresponding original real identity or PPI.

Moreover, some asthma medicines may require the principal to be of legal age. The agent then asks a RIdP for a RPI containing a pseudonym (e.g., a random number) and containing an attribute that states that the agent’s principal is of legal age. The RIdP can check this by verifying the birth date in the real identity of the agent’s principal. The agent can show this attribute when purchasing medicines that require being of legal age and concealing this attribute otherwise (e.g., when purchasing medicines for a cold).

Moreover, seller agents could construct a detailed profile on the medicines needed by the principal. This allows seller agents to practice price discrimination. For instance, seller agents could infer that the buyer agent periodically purchases such medicines. Thus, they could charge a slightly increasing cost for each new transaction. The principal can instruct her/his buyer agent to use a different new RPI each time it purchases asthma medicines in order to avoid this. Thus, it is difficult for a seller agent to be aware that different transactions were performed by the same buyer agent under different RPIs.

Buyer agents are able to choose among seller agents that sell the same medicines. One of the important dimensions that buyers will take into account in their decisions is the trust that they have in each seller agent. This trust can be based on successful previous interactions with the same seller agent. A buyer agent can trust in a seller agent in regard to past interactions by measuring: whether or not the seller agent shipped the product in time, the overall quality of the product bought, if there were hidden costs, etc. If the buyer agent has no previous interactions with a seller agent, the buyer agent can also consider the reputation of the seller agent in the marketplace.

In this scenario, identity-related vulnerabilities are a great concern. Seller agents should not be able to get rid of their trust and reputation ratings. This could cause important money loss. For instance, a seller agent could be cheating buyer agents by shipping medicines with a quality that is lower than expected. This obviously decreases the trust and reputation that buyer agents have in this seller agent. Hence, this seller agent decides to quit the electronic market and to reenter it with a new fresh partial identity, restarting its trust and reputation ratings from scratch. Another example would be a seller agent that sells the same medicine under different partial identities. This way, the probability of a buyer agent choosing one of its partial identities as the seller of the product increases.

\(^{10}\) We assume that payments are carried out using some kind of anonymous payment mechanism and that deliveries are carried out using some anonymous delivery system. Hence, credit card numbers and delivery addresses do not need to be disclosed when an agent acquires a product. For instance, the untraceable electronic cash presented by Chaum et al. [7] can be used for anonymous payments. For anonymous deliveries, the privacy-preserving physical delivery system presented by Aïmeur et al. [1] can be used.
If a buyer agent (and by extension its principal) wants to avoid identity-related vulnerabilities, it should only consider seller agents with a permanent partial identity (PPI). Thus, the buyer agent can use its own trust and reputation machinery to model the trustworthiness of these sellers and be sure that whitewashing and sibyl attacks are avoided.

Finally, accountability also needs to be considered. For instance, there may be seller agents that sell medicines illegally. For these cases, the real identity of the principal behind a seller agent that sells medicines illegally can be known. A court could require the PiDp to disclose the real identity behind a PPI. As a result, the principal holding this real identity could be sued for selling medicines illegally. The final punishment may depend on the applicable laws for such a case.

7 Conclusions

In this paper, we present the privacy-enhancing support that Magentix2 provides. This privacy-enhancing support also avoids identity-related vulnerabilities of trust and reputation models as well as the lack of accountability of the principals involved. All these features are crucial for encouraging principals’ trust in agent-based e-commerce applications.

Agents running on Magentix2 can use these features at will depending on their principals’ needs. An agent can create as many RPIs as needed to avoid information processing. Otherwise, an agent can use a PPI if it is interested in building trust and reputation. Thus, other agents can trust in this agent while being sure that it cannot perform whitewashing and sibyl attacks.

As future work, we plan to explore the possibility of agents with advanced reasoning capabilities deciding whether or not to use a new RPI for a given transaction based on possible privacy risks. These agents will be implemented using the Magentix2 native support for BDI agents. To this aim, the Jason framework [5] has been integrated into Magentix2.

References


Auto-adaptation of Open MAS through On-line Modifications of the Environment*

Roberto Centeno¹ and Holger Billhardt²
rcenteno@lsi.uned.es holger.billhardt@urjc.es

¹ Departamento de Lenguajes y Sistemas Informáticos
Universidad Nacional de Educación a Distancia, Madrid, Spain
² Centre for Intelligent Information Technology (CETINIA)
University Rey Juan Carlos, Madrid, Spain

Abstract. In this paper we propose a mechanism that is able to encourage agents, participating in an open multiagent system, to follow a desirable behaviour, by introducing modifications in the environment. This mechanism is deployed by using an infrastructure based on institutional agents called incentivators. Each external agent is assigned to an incentivator that is able to discover its preferences, and to learn the suitable modifications in the environment, in order to encourage its agent to perform actions, such that, the system’s preferences are fulfilled. Finally, we evaluate the mechanism in a p2p scenario.

Keywords: Incentives, Regulation, Adaptation, Organisation

1 Introduction

Open MultiAgent Systems (OMAS) are systems designed with a general purpose in mind but with an unknown population of autonomous agents at design time. The agents that populate such systems may be heterogeneous (e.g., they may have different unknown preferences and behaviour) and their number may vary dynamically at runtime. Based on their open nature, the general problem when designing OMAS consists in assuring that agents will behave according to the system’s global objectives and preferences.

The research community has tackled this problem by defining organisational models that structure and regulate the agent’s action space. Some approaches (e.g., the one adopted in Electronic Institutions [6]), define the actions that agents can take in each state of the system (performative structure) and rely on a certain infrastructure that assures that agents cannot violate the defined rules. In these proposals, the implemented rules – defined at design time – are fixed. They can be seen as heuristics that help to meet the system’s global objectives. This approach has two disadvantages. First, the agents may still have a certain degree of freedom and this may still imply a more or

* The present work has been partially funded by the Spanish Ministry of Education and Science under projects TIN2009-13839-C03-02 and “Agreement Technologies” (CONSOLIDER CSD2007-0022, INGENIO 2010)
less efficient completion of the systems overall objective. Second, the fixed nature of predefined rules may imply less flexibility in certain unforeseen situations. This may occur especially in highly dynamic and complex systems.

Alternative approaches (e.g., OMNI [4]) deal with this problem. They also define the valid actions in terms of norms, but agents are able to violate such norms [4]. In order to avoid violations those approaches rely on penalties/rewards and implement violation detection mechanisms. However, a new problem may emerge, what happens if the current population of the system is not sensitive to the defined penalties and rewards?

In our opinion, in (norm based) OMAS it is hard to specify a good set of norms at design time. It may not be clear whether the proposed punishments/incentives have the desired influence on the agents nor whether the specified norms may actually effect the global utility in a positive way. Addressing the aforementioned problems, we propose to endow OMAS with a mechanism that tries to induce agents at each moment to act in a way that is appropriate from the point of view of the global utility of the system. The work is inspired by the theory "Economic Analysis of Law", proposed by R.A. Posner in [13]. In this work the author analyses normative systems from an economic point of view. He focuses on the effects of norms in terms of outcomes on both, the behaviour of individuals and the society as a whole. Assuming that individuals are rational, norms are actually incentives that may induce agents to act in a certain way and this, in turn, may have a certain effect on the society. Following these ideas, we propose an adaptive incentive mechanism that: i) is able to identify the appropriate agents' actions from the systems point of view, ii) estimates agents' preferences, and iii) induce agents to act in the desired way by modifying the consequences of their actions.

The paper is organized as follows, Section 2 provides basic definitions and assumptions. Our approach is presented in Section 3. Section 4 describes the experimental validation we have carried out in a p2p scenario. Section 5 puts forward some related work. Finally, Section 6 gives some conclusions and points out lines of future work.

2 Definitions and Assumptions

We model an agent participating in a open multiagent system (OMAS) as a rational utility maximizer defined as a tuple \( \langle S, g, U, t, s_0 \rangle \); where \( S \) is the set of internal states of the agent (\( s_0 \) is the initial state); \( g : X' \times S \rightarrow S \) is the agent’s state transition function that assigns a new internal state to the current state and a partial observation of an environmental state \( x' \in X' \); \( U : S \rightarrow \mathbb{R} \) is the utility function that assigns a value to each possible internal state; and \( t \) is the agent’s decision function such that \( t : S \rightarrow A \) follows the principle of maximising the expected utility (MEU). That is, \( t(s) = \arg\max_{a \in A} eu(a, s) = \arg\max_{a \in A} \sum_{s' \in S} U(s') \cdot P(s'|s, a) \), where \( A \) is the set of possible actions; \( eu(a, s) \) is the expected utility of performing the action \( a \) in the state \( s \); \( U(s') \) is the utility of the state \( s' \) estimated by the agent; and \( P(s'|s, a) \) is the agents’ estimate, at state \( s \), of the probability that state \( s' \) will occur when executing action \( a \) in state \( s \).

\footnote{In this work we assume that agents have a partial but perfect observation of the environment.}
Agents participate in an OMAS that specifies the environment in which they carry out their activities. An OMAS is modelled as a tuple \( \langle A_g, A, X, \Phi, \varphi, U, x_0 \rangle \); where \( A_g \) is a set of agents, \( |A_g| \) denotes the number of agents in the system; \( A \) is a possibly infinite action space that includes all possible actions that can be performed in the system; \( X \) is the environmental state space; \( \Phi : X \times A^{\vert A_g \vert} \times X \rightarrow [0..1] \) is the OMAS transition probability distribution, describing how the environment evolves as a result of agents’ actions; \( \varphi : A_g \times X \times A \rightarrow \{0, 1\} \) is the agents’ capability function describing the actions agents are able to perform in a given state of the environment (physical restrictions); \( U : X \rightarrow \mathbb{R} \) is the utility function of the system that assigns a value to each environmental state; and, finally, \( x_0 \in X \) stands for the initial state.

OMAS are usually designed with a general purpose in mind – the global objective of the system. This objective is represented by means of preferences which are captured by the utility function \( U \) of the system. From the point of view of the designer, the problem consists of how to optimise the global utility of the system assuming that agents will try to optimise their own individual utilities. In order to do this, the agents’ behaviour may be influenced by endowing the system with some kind of organisational mechanisms\[2\]. In particular, in this work we focus on incentive mechanisms.

Our notion of “incentive” is slightly different to the usual consideration of incentives to be something positive. In our work, we consider that incentives are modifications of the environment that have the aim to make a particular action \( a \) more attractive than other alternatives \( \{b, \ldots\} \) and such that a rational agent would decide to take \( a \). This can be done either by making action \( a \) more attractive or by making the alternatives less attractive. In the framework of the given definitions above, an incentive mechanism is a function \( Y_{inc} : X' \rightarrow [X \times A^{\vert A_g \vert} \times X \rightarrow [0..1]] \) that taking into account its partial view of the environmental state, changes the transition probability distribution \( \Phi \) of an OMAS. The rationale behind this definition is that changing the consequences of actions may produce variations in the expected utility of agents. Therefore, they would change their decisions accordingly to the new consequences. We say that an incentive mechanism is effective if its implementation implies an improvement of the utility of the system \( U \).

We make the following assumptions:

**Assumption 1** The action space in the system is finite.

**Assumption 2** the environment of a system can be discretized by a finite set of attributes: \( X = \{X_1, \ldots, X_n\} \). An environmental state \( x_i \in X \) can be modelled as a set of tuples \( x_{i,j} = (attribute, value) \) assigning a value to each attribute. The incentive mechanism has permission to modify the values of at least some of those attributes.

**Assumption 3** agents have multi-attribute utility functions (\[10\]) defined over some attributes of the environment. The attributes are additively independent. That is, the utility function can be expressed as: \( U(s) = \sum_{j=1}^{n} w_j \cdot u_{i,j} \) where \( u_{i,j} \) is the utility of the attribute \( x_{i,j} \) perceived in the state \( s \) and \( w_j \) is the weight of attribute \( X_j \). Their preferences do not have to be aligned with the preferences of the system.
Assumption 4 The global utility function of the OMAS is a multi-attribute utility function over the attributes that define an environmental state and where attributes are additively independent.

3 Effective Incentive Mechanism

The proposed incentive mechanism accomplishes two basic tasks: i) selecting the actions to be promoted in order to improve the global utility of the system; and ii) performing the required changes in the environment in order to make the desired actions more attractive for the agents.

Whereas in standard normative approaches both tasks are solved at design time (by specifying norms), we propose to solve them at runtime. We use learning algorithms that allow to adapt the incentive mechanism to the current agent population as well as to possible changes in the environment.

Similar to the use of an infrastructure in other organisation-based multi-agent systems for regulating the interactions between external agents and the system (e.g., in AMELI in Electronic Institutions [7]), we propose to deploy the incentive mechanism as an infrastructure. We propose to endow an OMAS with institutional agents, called incentivators, in charge of implementing the functionality of the incentive mechanism for a particular agent. That is, each external agent is assigned to an incentivator that aims to discover agent’s preferences, as well as, to learn how to stimulate it by modifying the consequences of its actions. Furthermore, incentivators can communicate with each other allowing them to coordinate their actions.

3.1 Discovering Agents’ Preferences

The action selection mechanism of rational agents is based on an ordering over actions, in terms of expected utility. In order to induce such agents to perform a certain action, an incentive mechanism could either modify the consequences of that action in such a way that the resulting state is more attractive, or it could modify the consequences of the rest of the actions in order to make them less attractive. If the agent’s preferences are known, this becomes a relatively easy task. However, in open systems, agents’ preferences are unknown and, thus, have to be discovered.

One approach to preferences elicitation is either to ask agents a set of questions regarding their preferences. Based on the identified preferences, the utility functions can be estimated [1]. However, this approach has obvious disadvantages, especially in non-collaborative domains, e.g. agents could lie giving their answers. Alternatively, we propose to use a non-intrusive approach where each incentivator discovers its agent’s preferences by observing its behaviour in response to given incentives. The characteristics of the discovering process are: i) it is a learning process; ii) it is independent, i.e., the incentivator does not require to coordinate with any other incentivator; and iii) the incentivator receives an immediate local reward, i.e., whether the agent reacts to the modifications in the environment or not. With these characteristics in mind, Q-learning with immediate rewards and $\epsilon$-greedy action selection [15] has been chosen as mechanism to carry out this task. In each step, each incentivator selects the most promising
attribute to modify and a value for this attribute, applies the changes, observes its agents
reaction and modifies the q-values for attributes and values accordingly.

For the sake of simplicity, the process has been split up into two different but related
learning process. On one hand, the incentivator has to discover the attributes that affect
an agent’s utility function, and, on the other hand, it has to identify the required values
of these attributes in order to make an agent change its mind.

**Learning the attributes that influence agents’ behaviour** In the scope of Q-learning,
the action space $Z_i$ of the incentivator for agent $ag_i$ is composed of the attributes the
incentivator is authorised to modify in the system. More formally: $Z_i \subseteq \{X_1, \ldots, X_n\}$,
where $X_j$ are attributes belonging to the environmental state of the system. Thus, when
the incentivator takes the action $z_{i,j}$ this means that the attribute $X_j$ is modified. After
that, it receives a reward that rates that action, and the action-value function estimation
is updated as follows:

$$Q_{t+1}(z_{i,j}) = Q_t(z_{i,j}) + \alpha \cdot [R_t(z_{i,j}) - Q_t(z_{i,j})]$$

(1)

where $\alpha$ is the learning rate and $R_t(z_{i,j})$ the reward. As we said before, the idea is
to discover an agent’s preferences by observing how it reacts to the modification pro-
posed. Thus, an action is rated positively if the external agent performs the action the
incentivator wanted to, and negatively if not. This is captured by using the following
reward function:

$$R_t(z_{i,j}) = \begin{cases} +1 & \text{if agent performed the action} \\ -1 & \text{i.o.c.} \end{cases}$$

(2)

Besides, in order to explore new attributes, a random selection is made with small prob-
ability $\epsilon$, and the highest q-value attribute (greedy action) is exploited (chosen in the
next step) with probability $(1 - \epsilon)$.

**Learning the values of attributes** The next step in the learning process is to learn the
most effective value of the attribute selected. The characteristics of this problem are the
same as the attribute learning process. Again, Q-learning with immediate rewards and $\epsilon$-greedy action selection is used. In this case, the action space $Y_{i,j}$ of the incentivator
for agent $ag_i$ depends on the attribute $X_j$ selected previously by the attribute learning
algorithm. It is composed of the different values that $X_j$ may take. Formally: $Y_{i,j} = \{value \in [value_{min}^{X_j}, value_{max}^{X_j}]\}$, where value stands for the set of different values
the attribute $X_j$ may take. As update and reward functions we use the same formulae
as before (equations 1 and 2).

Combining both learning phases, in each step an incentivator proposes a modifica-
tion – a new value for a pair $x_{i}^*=\langle\text{attribute, value}\rangle$. Over time, $x_{i}^*$ eventually converges
towards a pair that influences the behaviour of agent $ag_i$.

---

4 This approach requires the set of values to be discrete. In case it is continuous, it should be
discretized previously.
3.2 Identifying desirable actions

As we have introduced previously, the incentive mechanism has to decide which actions should be incentivized in order to improve the system’s utility. In scenarios where the outcome of the action performed by an agent does not depend on the actions taken by others, these actions could be determined locally. However, in many common situations the outcome of an action depends on the joint action of all participating agents. In order to account for this fact, all incentivators should work as a team so as to coordinate the actions to be promoted. The main problem in order to carry out such a task is that incentivators have just a local view of the system – the result of the action performed by “their agents”.

Therefore, this task should have the following capabilities: i) learning a joint action in a cooperative way; ii) dealing with the lack of information about the actions taken by other members of the team; and iii) dealing with immediate local rewards. With this in mind, incentivators are endowed with a reinforcement multiagent cooperative learning algorithm that updates the action-value function estimation with the typical Q-learning function (equation 1). As reward function we use the global utility, that is calculated by aggregating the local rewards through a gossip-based algorithm.

Incentivator’s action space The first issue that needs to be addressed, is determining the action set of an incentivator in this learning task. It is composed of the set of actions its agent ($a_{gi}$) can take in the current situation, combined with the attribute modification selected by the attribute learning algorithm described earlier. Formally: $V_i \subseteq \{\emptyset, \beta_{i,1}, \ldots, \beta_{i,n}\}$; where $\emptyset$ is the skip action; and $\beta_{i,j} = \langle a_{i,j}, x^{*}_{i,j}\rangle$, where $a_{i,j}$ stands for an action agent $a_{gi}$ can perform in the current situation; and $x^{*}_{i,j}$ is the attribute modification selected as result of the learning process explained in 3.1. When an incentivator takes the action $\beta_{i,j}$, this means that the action $a_{i,j}$ should be made more attractive by modifying its consequences through a change of the $\langle$attribute, value$\rangle$ pair $x^{*}_{i,j}$. The action $\emptyset$ means that none of the actions will be promoted (e.g. no parameters of the environment will be changes).

Calculating the reward After taking an action ($v_{i,j} \in V_i$), the incentivator $i$ receives a reward that rates such an action, and the action-value function estimation is updated by using the corresponding formula (equation 1). This reward is calculated as follows: $R_i(v_{i,j}) = \bar{U}_i(x)$; where $\bar{U}_i(x)$ is the incentivator’s estimation of the global utility in the environmental state $x$ reached after the last step. Since an incentivator has only a local view of the system, it can calculate $\bar{U}_i(x)$ only based on its local perception of the environment$^5$. In order to take into account the actions taken by other incentivators, it should transform its local estimation into an estimation of the global utility. Based on the assumption of additive independence of the attributes in the system’s utility function, the global utility can be estimated by aggregating the local utility estimations of all incentivators. In order to perform this task, each incentivator is endowed with the gossip-based aggregation algorithm presented in [9].

$^5$ We assume that incentivators know the system’s utility function.
The idea is that each incentivator holds a local value, and by exchanging messages with its neighbours the local values are aggregated by using some aggregation function. Two different threads are executed (see Table 1). The active thread periodically initiates an information exchange with a random incentivator \( j \) by sending a message containing the local utility estimation \( \bar{U}_i(x) \) and waits for a response with the utility estimation \( \bar{U}_j(x) \) from incentivator \( j \). On the other hand, the passive thread waits for messages sent by other incentivators and replies with the local utility estimate. The update method updates the local utility estimation by aggregating the current value and the received value. In our particular use case (see Section 4), we have chosen the average. Therefore, \( \text{update}(\bar{U}_i(x), \bar{U}_j(x)) \) returns \( (\bar{U}_i(x) + \bar{U}_j(x))/2 \). This function decreases the variance over the set of all local estimates of the global utility.

### 3.3 Interaction between agents and incentivators

Two different types of interactions could be performed between an incentivator and its agent. On one hand, in order to enable agents to reason about incentives, the incentivator informs its agents about the consequences of the desirable actions. Before an agent selects a new action, it will query the incentivator asking for the possible new consequences of the actions the agent considers\(^6\).

On the other hand, each incentivator observes the reaction of “its” agents to the proposed incentives. This information provides feedback for the preference learning algorithm described earlier. It should be noted that it is possible that an agent may perform an action because of its own interests (and not because of the proposed incentive). We do not have any mean to distinguish such situations. We assume that the exploration/exploitation process will detect such situations and converge to an estimation of the agent’s preferences.

### 4 Case Study: a P2P System

We have chosen a peer-to-peer (p2p) file sharing scenario for evaluating our approach. Such scenarios are clear examples of open systems where the objectives of individuals may not coincide with the objectives of the system.

In such systems normally only few peers (seeders) have the whole information; and the rest of peers (leechers) download pieces by using a particular protocol. We focus

---

\(^6\) We suppose rational agents will always use this “service” since it is in their own interest.
on peers sharing a file with the BitTorrent protocol [3]. Following this protocol, a file is split in pieces of 256KB each and every piece is split in 16 sub-pieces called blocks. Peers exchange blocks and each block can be downloaded from different peers. For the sake of simplicity we leave out the phases in which peers and data are identified and peers get a list of neighbours to communicate with (view figure 1). We just focus on the phase carried out to get each block. In this phase, each peer sends a bitfield message to its neighbours asking for their file status. After that, the peer has to decide which neighbours will be asked for the next block to download. A request message is sent to the selected peers. When a peer receives a request message it has to decide whether the requested block will be uploaded or not. Once a peer accepts the request, it sends a piece message containing the requested block. Immediately, the receiver of the piece sends a cancel message to the other neighbours it asked for the same block. When the download is finished, a have message is sent to the agent’s neighbours in order to update their file status.

Fig. 1. Simplified BitTorrent Protocol

4.1 P2P system model

Regarding the most common problems in p2p systems (e.g. non-cooperation of peers, flooding of the network, etc. [8]), the objectives of the system could be specified as follows: i) peers should download/upload as many files/blocks as possible in order to increase the number of available files; ii) the usage of the network should be as low as possible; and iii) the time spent on downloading files should be as short as possible in order to avoid an overload of the network. These objectives might be captured by a multi-attribute utility function as follows:

\[
U(x) = U_{files} \cdot w_0 + U_{blocks}(x) \cdot w_1 + U_{cn}(x) \cdot w_2 + U_{ct}(x) \cdot w_3
\]

where, \(U_{files}(x)\), is the utility of the number of already downloaded files, \(U_{blocks}(x)\) represents the utility of the number of downloading/uploading blocks in the state \(x\), the greater the number of downloading/uploading blocks, the greater the utility is; \(U_{cn}(x)\) is the utility of the usage of the network in the state \(x\), following the work presented in [11], we have defined this parameter as network cost and it represents the sum of the network usage of each message \(c_{mi}\) sent among agents. It is calculated as follows:
\[
C_n = \sum_{i=0}^{\#msgs} c_{m_i} \text{ such that } c_{m_i} = m_{\text{length}} \cdot \text{Lat}(m_{\text{org}}, m_{\text{dst}})
\]

where \(m_{\text{length}}\) is the length of a message, and \(\text{Lat}(m_{\text{org}}, m_{\text{dst}})\) is the latency between the origin and destiny of the message\(^7\). The lower the network cost, the greater the utility is. Finally, \(U_{ct}(x_j)\) is the utility of the time spent on downloading a file (time cost). The shorter the time cost, the greater the utility of the system. The partial utilities \(U_x\) are calculated, in case of maximization, as the ratio between the actual value of the parameter \(x\) divided by the maximum possible value of the current state (1 minus the ratio in case of minimization). \(w_0, w_1, w_2, w_3\) allow us to weight each attribute.

4.2 Peer agent model

Peers are modelled as rational agents that follow their own preferences. We focus on two main decisions peers have to make: i) to decide to how many neighbours it will send a request message asking for the next block to download; and ii) to decide how many requests received from other peers are accepted (i.e., how many piece messages are send). Accordingly, the action space of a peer is: \(A = \{\text{sendPiece}(n), \text{sendRequest}(n), \text{skip}\}\), where \(n\) is the number of neighbours it sends a piece or request message\(^8\).

Peers that join the system obtain a certain bandwidth. Furthermore, we assume that peers have to pay a regular fee in order to connect to the network. Besides, a peer has a file it is sharing and whose status can be partially/completed downloaded. When a peer joins the system it receives a list of peers (neighbours) it can contact. We assume that a peer knows the latency with all its neighbours\(^9\); and their file status is updated when have messages are received.

The attributes that may have some influence on a peer’s preferences are bandwidth, fee, number of downloading/uploading blocks of a file and time spent on downloading a file. We capture such preferences through the following general multi-attribute utility function:

\[
U(x) = U_{bw}(x) \cdot w_4 + U_{fee}(x) \cdot w_5 + U_{down}(x) \cdot w_6 + U_{up}(x) \cdot w_7 + U_t(x) \cdot w_8
\]

where \(U_{bw}(x)\) is the utility of the bandwidth rate - the bandwidth regarding to the available bandwidth; \(U_{fee}(x)\) represents the utility of the fee that has to be paid to connect to the network; \(U_{down}(x)\) stands for the utility of the number of downloading blocks; \(U_{up}(x)\) is the utility of the number of uploading blocks to other peers; \(U_t(x)\) represents the utility of the time spent on downloading a file; and \(w_4, w_5, w_6, w_7, w_8\) are the weights assigned to each attribute.

Different kind of peers can be modelled by instantiating the weights of this utility function. For instance, we can model agents interested in having the greatest number of downloading blocks, or agents interested in having the lowest available bandwidth possible, etc.

---

\(^7\) Latency is assumed to be constant and symmetric.

\(^8\) Note that the action that, for instance, sends two requests to neighbours 1 and 2 is different to the one that sends two requests to neighbours 1 and 3.

\(^9\) It can be easily calculated by using ping messages.
4.3 Regulating the system

With the aim of improving the system’s global utility, we have endowed the system with two different types of regulation mechanisms: a standard normative system, and our incentive mechanism.

The normative system is based on a set of norms coupled with penalties that are applied when norms are violated. In particular, three norms have been designed at design time, that is, before knowing the population of the system: N1: "It is prohibited to use more bandwidth than $x\%$"; N2: "A peer is obliged to upload a block when at least $y\%$ of the bandwidth is available"; and N3: "It is prohibited to request a block to more than the $z\%$ of neighbours". The set of norms is designed according to the global objective of the system. Norm violations are penalised with an increase of the fee in 5 units. Such violations are detected with a 100% efficiency.

Regarding the incentive mechanism, it is deployed by taking advantages of the own nature of p2p systems. That is, incentivators are located at network service providers. Thus, the communication among them will be fast. Incentivators are authorized to modify the bandwidth assigned to peers and the fees peers pay to connect to the network. So, the action spaces agent $a_{gi}$'s incentivator are $Z_i = \{fee_{a_{gi}}, bw_{a_{gi}}\}$ and $Y_{i,fee} = \{value_{fee_{a_{gi}}} \in [value_{min_{fee_{a_{gi}}}}, value_{max_{fee_{a_{gi}}}}]\}$ in case the attribute selected is the fee; or $Y_{i,bw} = \{value_{bw_{a_{gi}}} \in [value_{min_{bw_{a_{gi}}}}, value_{max_{bw_{a_{gi}}}}]\}$ in case of bandwidth. The actions that can be incentivized by incentivators are sendPiece(n), sendRequest(n) and skip, where n can be instantiated by a number of neighbours.

4.4 Experimental results

We have conducted some experiments to evaluate our proposal and to compare it to other approaches. We have used the following setup: we have instantiated the p2p system, where peers only have to exchange a single file of 15 blocks. The latencies among all peers are randomly distributed in the range [10, 400] ms and the bandwidths in [640, 1024, 2048, 4096] Mb/s. The fee an agent has to pay to connect the system is randomly selected in the range [10, 50] with steps of 5 units. In order to calculate the network cost measure of the system a message length for each kind of message is required. The following values have been assigned: piece message = 128Bytes, request and cancel message = 1Byte, and incentivators communication = 1Byte. The simulation environment has been developed at a synchronous way, that is, the execution is performed by time steps. Each time step is equivalent to 10ms, it means that a request/cancel message sent between two peers connected with a minimum latency of 10ms takes 1 time step to arrive. Moreover, each message has a ttl of 6 time steps since it arrives to the destiny, that is, a request, not answered in less than 6 time steps, will be discarded. The utility function of the system is weighted with the following parameters: $w_0 = 0.5$, $w_1 = 0.2$, $w_2 = 0.125$ and $w_3 = 0.175$. Regarding the normative system, norms were initiated with $x = 85\%$, $y = 25\%$ and $z = 85\%$. The learning algorithms in the incentive mechanism are initialised with $\alpha = 0.9$ and $\epsilon = 0.1$. Besides, the q-values are initialised optimistically with 1 (the maximum value) in order to assure more exploration at the beginning.

$x$, $y$, $z$ are percentages that can be defined by the designer.
Experiment 1: Non collaborative agents that are sensitive to the fee  In this experiment the p2p system is populated with 50 peers. They are modelled as non collaborative agents. That is, they are interested in downloading blocks, but not in uploading. In addition, they are sensitive to modification in the fee they are paying for connecting to the network. This is captured by the following weights in their utility function: $w_4 = 0.2$, $w_5 = 0.399$, $w_6 = 0.3$, $w_7 = 0.001$ and $w_8 = 0.1$. Each peer receives a list of 2-4 neighbours that is randomly generated among all possible peers. Moreover, a 25%, as maximum, of peers are seeders, and the rest of them are leechers. All peers have, at least, 1 seeder as neighbour.Leechers have already downloaded between 0-14 blocks, that is, they aim to download between 1 and 15 blocks. In the normative system, norm violations are penalised with an increase of the fee in 5 units.

The experiment has been executed with four different configurations: i) without any regulation, ii) with the normative system, iii) with our proposal, and iv) with both, normative and incentive mechanisms at the same time. Figure 2(a) plots the average utility obtained by all peers participating in the system. As we can see, agents obtain the highest utility when the system is executed without any regulation mechanisms. That is obviously, because nothing restricts the freedom of agents to do what they want. The second best performance is of the incentive mechanism. The reason is that agents are regulated by giving incentives, that is, they are incentivized to perform in a cer-
tain manner by paying a lower fee, in this particular case. The normative system and the combination of both mechanisms work similarly "bad" regarding the agents utility. That happens because norms restrict agents’ behaviour, so, their utility is lower because of the punishments. Regarding the system, figure 2(b) plots the evolution of the its utility for each configuration. It shows clearly that the systems utility is low if no regulation takes place. This was expected because the analysed population of peers does not behave according to the system’s preferences. On the other hand, the normative and incentive mechanisms work similarly well. The incentive mechanism is a bit “slower” at the beginning due to the learning algorithm, that is, incentivators have to discover that the money is the attribute that has influence on the peers. Finally, the combination of incentive and normative is the one that performs best. It could happen because norms restrict the valid actions and the incentive mechanism induces agents to perform the desirable actions, so utility is always improved. This is also shown in figure 2(c), which compares the four mechanisms regarding to the number of peers that are able to download the whole file and the time spent on it. By using the combination of incentive and normative, all peers (50) are able to download the whole file. On the other hand, 49 peers are able to download the file when using the incentive and the normative mechanisms respectively. However, the time spent is higher in case of the incentive mechanism, due to the learning process, as it has been pointed out previously.

Concluding, in the case of previous knowledge about the agents preferences and where norms have been designed by taking into account such information, the incentive mechanism works similar to the normative system.

Experiment 2: Non collaborative agents that are not sensitive to the fee  In this case the system is populated by 50 peers that are not collaborative as well. This is captured by the following weights in their utility function: \( w_4 = 0.2, w_5 = 0.001, w_6 = 0.399, w_7 = 0.001 \) and \( w_8 = 0.399 \). We have specifically chosen a peer population that is not sensitive to changes in the fee they are paying, e.g., they do not care about increasing fees. This can be seen from the weights in the agents’ utility function (in particular \( w_5 = 0.001 \)). This describes a case where the designed norms (all based on an increase in the fee as a punishment) will not be very effective for the given population of agents. Figure 3(a) plots the average utility obtained by all peers. The results are clearly better when the incentive mechanism is employed, either alone or coupled with the normative system. The results are clearly better when the incentive mechanism is employed. Implicitly, this mechanisms is able to identify that instead of the fee, the bandwidth has an influence on peers’ utility. It uses changes in the available bandwidth to make the upload of blocks to other peers attractive. Regarding the system, figure 3(b) plots the utility of the system when it is regulated by the different mechanisms. As it was expected, the system improves its performance when it is regulated by the incentive mechanism because it is able to influence agents’ behaviour regarding the sending of their blocks. Finally, in figure 3(c) we can see the number of peers that are able to download the whole file. In the case of normative and no mechanisms, none of the peers are able to download the files, only the initial seeders have the whole file, that is why the time spent on downloading the files is zero. On the other hand, with the incentive mechanism 48 out of 50 peers download the whole file, by spending 7640ms.
In contrast, when the incentive mechanism is coupled with the normative system the results are improved and 49 out of 50 peers download the whole file spending less time (6770ms). This may be due to the fact that the influence of the fee on the utility of agents is not zero, and thus, the punishments together with the incentives may have a better behaviour.

It is important to note that the overhead introduced as a consequence of the gossip algorithm is taken into consideration in the network cost attribute calculated in the system utility function. That means, the incentive mechanism is beneficial even if the overhead is taken into account. The experiment makes clear that a normative system does not fulfil its goal if the agents are not sensitive to the applied punishments. In contrast, the proposed incentive mechanism is able to adapt to such situations.

5 Related Work

Many approaches have been proposed, in the field of multi-agent organisations, to regulate the activities of agents in a MAS [6, 4]. In most of them, the concept of norm appears as a main piece. Some approaches, like [6], focus on defining the set of allowed actions in each possible state of the system and assure that agents are just allowed to perform the valid actions. We think our approach is compatible with such approaches. Our work can be used for inducing a particular action, among all possible valid actions, optimising in that way the system’s efficiency. Other approaches, like [4], propose to
couple norms with penalties/rewards which give agents the possibility to violate a norm. In these approaches norms are usually defined at designed time, what implies that some assumptions are made with regard to the agent population (e.g. the attributes that have an influence on agent’s preferences). In our work, we do not assume a priori knowledge about agents’ preferences; our approach tries to learn such preferences for each agent and, thus, provides a basis for an individualised incentivation/punishment mechanism.

Our work is quite related to mechanism design [12], as it aims at influencing the behaviour of rational agents in a desired direction. However, instead of directly defining the (interactions) rules of the system, we try to achieve desired behaviour of agents by modifying their environment at runtime (i.e. agents interact only through environment). Furthermore, we do not assume that the agents’ pay-off functions are known, but we intend to learn which of the environment attributes are relevant for an agent’s pay-off, so as to dynamically adjust incentives for it at runtime.

Recently, some papers have been published with a similar focus as ours [14, 5, 16, 17]. All of them address the same problem, how to influence agents’ behaviour in order to induce some desirable behaviour. As solution they focus on formal approaches based on environment design techniques, incentive based approaches and behaviour cultivation. They prove by experimental results that their approaches effectively induce desirable behaviour. However, they do not deal with scalability issues. Thus, the main contribution of our work regarding to them, is that we propose to deploy the mechanism by using an infrastructure where the control and computation is distributed.

Regarding p2p systems, in [11] the authors propose to regulate and adapt a p2p system by learning and establishing new social conventions with the aim of improving the system’s performance. The main difference to our work is that they assume a certain control over agents, because when a new convention is learnt, it is established and assumed that agents will follow it. Other works have proposed incentive-based solutions for p2p related problems (e.g., [8]). However, it is usually assumed that some attribute have an influence on peers’ preferences and punishments and incentives are defined based on this assumption.

6 Conclusion

In this paper we have put forward an incentive mechanism that is able to induce desirable behaviour by modifying the consequences of actions in open MAS. The mechanism tries to discover which attributes have some influence on agents’ preferences and learns how agents can be incentivized. It is deployed by using an infrastructure based on institutional agents called incentivators. The incentivators use the Q-learning algorithm to discover agents’ preferences by observing how they react to modifications in the environment. Moreover, they learn – in a cooperative way, using Q-learning and a gossip-based reward calculation algorithm – which joint actions should be incentivized in order to increase the utility of the system. Finally, the proposed mechanism has been tested in a p2p file sharing scenario, showing that it is a valid alternative to standard normative systems. In particular it outperforms regulation mechanisms based on fixed norms if the design assumptions of such norms are not fulfilled.
As future work, we plan to explore other learning techniques that may be more suitable in scenarios where agents preferences may vary over time. In principle, Q-learning can deal with such situations, but it may be too slow to obtain the desired adaptation of the system. Furthermore, the mechanism is currently able to perform the modification of just one attribute at the same time. In this sense, we will explore the possibility of modify a set of attributes jointly.

References

Combining Semantic Web and Logic Programming for Agent Reasoning

Murat Şensoy, Wamberto W. Vasconcelos, and Timothy J. Norman

Department of Computing Science, University of Aberdeen, AB24 3UE, Aberdeen, UK
{m.sensoy,w.w.vasconcelos,t.j.norman}@abdn.ac.uk

Abstract. Web Ontology Language (OWL) provides means to semantically represent domain knowledge as ontologies. Then, ontological reasoning allows software agents to effectively share and semantically interpret the knowledge. OWL adopts open world semantics and in order to achieve decidability its expressiveness is strictly limited. Therefore, many real-life problems cannot be represented only using ontologies and cannot be solved using just ontological reasoning. On the other hand, traditional reasoning mechanisms for autonomous agents are mostly based on Logic Programming (LP) and closed world assumption. LP provides a very expressive formal language, however it requires domain knowledge to be encoded as a part of logic programs. In this paper, we propose Ontological Logic Programming (OLP), a novel approach that combines logic programming with ontological reasoning. The proposed approach enables the use of ontological terms (i.e., individuals, classes and properties) directly within logic programs. The interpretation of these terms are delegated to an ontology reasoner during the interpretation of the program. Unlike similar approaches, OLP makes use of the full capacity of both the ontological reasoning and logic programming. Using case-studies, we demonstrate the usefulness of OLP in multi-agent settings.

1 Introduction

The Semantic Web is defined as an extension of the current Web in which information is given well-defined meaning, better enabling software agents and people to work in cooperation. This is achieved using an infrastructure that combines a set of technologies as illustrated in Figure 1. Web Ontology Language (OWL) plays a significant role in the fulfillment of Semantic Web vision. Description Logic (DL) is a decidable fragment of First Order Logic (FOL) [4]. It constitutes the formal background for OWL-DL, the decidable fragment of OWL [24]. However, DL is not sufficient on its own to solve many real-life problems. For example, some rules may not be expressed in DL. In order to represent rules in an ontology, rule languages such as Semantic Web Rule Language (SWRL) [1] have been proposed. In the design of Semantic Web languages, decidability has been one of the main concerns. To achieve decidability, these languages enforce limitations on expressiveness. OWL ensures decidability by defining its DL equivalent subset; similarly we can ensure decidability of SWRL using only DL-safe rules [10]. Existing reasoners such as Pellet [23] provide ontological reasoning services based on these restrictions. However, because of these limitations, many logical axioms and rules cannot be expressed using OWL-DL and SWRL [1].
On the other hand, languages like Prolog [25] provide very expressive declarative Logic Programming (LP) frameworks. Unlike OWL and SWRL, Prolog adopts the closed-world assumption\(^1\) through negation as failure and enables complex data structures and arbitrary programing constructs [25]. Many existing agent programming languages and reasoning mechanisms are based on LP [21]. Although LP provides a powerful framework for representation and reasoning, it does not have the benefits provided by Semantic Web, e.g., interoperability, knowledge reuse, and so on. In this paper, we propose Ontological Logic Programming (OLP)\(^2\), a novel approach that combines LP with DL-based ontological reasoning. An OLP program can dynamically import various ontologies and use the terms (i.e., classes, properties, and individuals) in these ontologies directly within an OLP program. The interpretation of these terms are delegated to an ontology reasoner during interpretation of the OLP program. By enhancing logic programming with ontological reasoning, OLP offers the following advantages:

1. **Expressiveness**: OLP combines the expressiveness of DL and LP. Hence, the limitations of OWL-DL are compensated by the high expressiveness of LP.
2. **Convenience**: Many researchers and developers are more familiar with LP languages than with DL formalisms. OLP enables DL reasoning to be used transparently within a logic program.
3. **Reuse of Domain Knowledge**: In logic programs, domain knowledge is encoded within the program, often in an ad-hoc manner. OLP enables domain knowledge to be defined in a set of ontologies in a standard way. These ontologies may then be easily used by different OLP programs.
4. **Conciseness**: OLP programs are far more concise than equivalent standard logic programs. This is because OLP programs use domain ontologies to reason about


\(^{2}\) OLP’s source code is publicly available at [http://olp-api.sourceforge.net](http://olp-api.sourceforge.net)
domain knowledge, while standard logic programs require this domain knowledge to be encoded within the program. Consider the rule *a person can drive a vehicle only if he/she has a driving license*. In a logic program, in order to express this rule, semantics and facts about the terms *person*, *transportation vehicle*, *driving*, and *driving license* have to be formalised within the logic program. This means that the program has to be much longer than the rule to be expressed. On the other hand, an OLP program simply imports appropriate ontologies that contain domain knowledge about these terms, and expresses the rule concisely using these terms.

5. **Reuse of Logic Programs**: Logic programming, notably Prolog, has been used for decades to develop many AI applications such as expert systems, planning systems, theorem provers, and so on. The proposed combination would allow “legacy” AI systems to take advantage of more recent Semantic Web developments, namely, open standards for knowledge representation with publicly available ontologies, as well as efficient reasoning mechanisms, without the need for re-implementation.

In this paper, we present Ontological Logic Programming in Section 2 both in terms of its architecture, and how OLP interacts with the underlying semantic knowledge. We then present two case-studies from the sensor resource management and team formation domains in Section 3. In Section 4, we discuss the contributions of our approach compared with key related research. Finally, we present our conclusions in Section 5.

## 2 Ontological Logic Programming

We present OLP in two stages. First, we introduce the OLP stack and describe how OLP interprets logic programs using semantic knowledge. We then discuss in detail how OLP modifies the underlying semantic knowledge and accesses semantic reasoning services.

![OLP Stack](image)

**Fig. 2. OLP Stack.**

### 2.1 Architecture

Figure 2 shows the stack of technologies and components used to interpret OLP programs. At the top of the stack, we have the OLP interpreter, which sits on top of a LP
layer. The LP layer is handled by a Prolog engine. The Prolog engine uses two different knowledge bases; one is a standard Prolog knowledge base of facts and clauses while the other is a semantic knowledge base composed of OWL-DL ontologies and SWRL rules. Pellet [23] has been used as a DL reasoner to interface between the Prolog engine and the semantic knowledge base.

Our choice of LP language is Prolog and in this work, we use a pure Java implementation, tuProlog [20]. The OLP interpreter is a Prolog meta-interpreter with a set of OLP-specific predicates, described in Section 2.2. Figure 3 shows a simplified version of the OLP interpreter used to evaluate OLP programs through the \texttt{eval/1} predicate.

While interpreting OLP programs, the system behaves as if it is evaluating a standard Prolog program until it encounters an ontological predicate. In order to differentiate ontological and conventional predicates, we use name-space prefixes separated from the predicate name by a colon, i.e., ".:”. For example, if W3C’s wine ontology \(^3\) is imported, we can directly use the ontological predicate \texttt{vin:hasFlavor} in an OLP program without the need to define its semantics, where \texttt{vin} is a name-space prefix that refers to \texttt{http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#}. This name-space prefix is defined and used in the wine ontology.

The Prolog knowledge base does not have any knowledge about ontological predicates, since these predicates are not defined in Prolog, but described separately in an ontology, using DL [4]. In order to interpret ontological predicates, the OLP interpreter needs ontological reasoning services provided by a DL reasoner. Hence, we have a DL reasoning layer below the LP layer. The interpreter accesses the DL reasoner through the \texttt{dl_reasoner/1} predicate as shown in Figure 3. This predicate is a reference to a Java method, which queries the reasoner and evaluates the ontological predicates based on ontological reasoning. OLP uses two disjoint knowledge bases. A Prolog knowledge base is used to store, modify and reason about non-ontological facts and clauses (e.g., rules), while a semantic knowledge base is used to store, modify and reason about ontological predicates and semantic rules. The semantic knowledge base is based on a set of OWL-DL ontologies, dynamically imported by OLP using \texttt{import} statements. Some rules are associated with these ontologies using SWRL [1]. Above the ontologies and the semantic rules, we have Pellet [23] as our choice of DL reasoner. It is used to infer facts and relationships from the ontologies and semantic rules transparently.

During the interpretation of an OLP program, when a predicate in \texttt{prefix:name} format is encountered, the DL reasoner below the LP layer in the OLP stack is queried to get direct or inferred facts about the predicate in the underlying ontologies. For example, when the meta-interpreter encounters \texttt{vin:hasFlavor(D,R)} during its interpretation of an OLP program, it queries the DL reasoner, because \texttt{vin:hasFlavor} is an ontological predicate. The \texttt{hasFlavor} predicate is defined in the wine ontology, so the reasoner interprets its semantics to infer direct and derived facts about it. Using this inferred knowledge, the variables \texttt{D} and \texttt{R} are unified with the appropriate terms from the ontology. Then, using these unifications, the interpretation of the OLP program is resumed. Therefore, we can directly use the concepts and properties from ontologies while writing logic programs and the direct and derived facts are imported from the ontology through a

\(^3\) It is located at \texttt{http://www.w3.org/TR/owl-guide/wine.rdf} and imports W3C’s food ontology located at \texttt{http://www.w3.org/TR/owl-guide/food.rdf}.
reasoner when necessary. In this way, OLP enables us to combine the advantages of logic programming (e.g., complex data types/structures, negation by failure and so on) and ontological reasoning. Moreover, logic programming aspect enables us to easily extend the OLP interpreter so as to provide, together with answers, explanations of the reasoning which took place.

Lastly, it is important to explain the effects of Prolog’s backtracking mechanism on our interpreter of Figure 3. The meta-interpreter undergoes backtracking in the standard fashion [3], exhaustively attempting to find a solution to a query $eval(G)$, trying different clauses in turn – the clauses, with the exception of $complex(G)$, are mutually exclusive, due to the patterns they have in their head goals. Prolog also tries different ways to prove the goals in the body of a clause, backtracking when one of them fails, and attempting to prove the previous goal again (hopefully obtaining a different set of values for its variables), until a solution is found to the last goal of the clause’s body.

We control the effects of backtracking on the invocation of the external DL reasoner, namely, the predicate $dl\_reasoner(O:G)$ in the first clause. We rely on the termination properties of our reasoner, Pellet, and the limited expressiveness of DL (for instance, circular definitions cannot be expressed), to compute all possible solutions for $O:G$ upon the first invocation of the predicate, and to produce these solutions one at a time upon backtracking.

2.2 Semantic Knowledge and OLP

OLP not only uses the semantic knowledge within ontologies, but it may also modify this knowledge by importing new ontologies, and adding or removing concepts, roles,
individuals and facts (i.e., RDF statements [24]). For this purpose, we provide OLP-specific predicates. Here, we outline how OLP may be used to modify the semantic knowledge base:

- **Importing ontologies.** In a classical Prolog program, domain knowledge is encoded as part of the Prolog knowledge base. To facilitate the reuse of standardised domain ontologies, OLP enables Prolog programs to directly use predicates defined in ontologies. An OLP program may import a number of ontologies to access the domain knowledge encoded within them. We provide two mechanisms to do this. First, at the beginning of an OLP program, lines starting with `%import` are interpreted as an instruction to import an ontology located at a specific URI (note that these lines start with `%`, so they are regarded as comments by the Prolog engine). Second, the `import_ontology` predicate can be directly used within an OLP program to dynamically import new ontologies.

- **Addition and removal of statements.** As shown in Figure 3, the OLP interpreter evaluates `assert` and `retract` predicates differently depending on whether these are ontological and non-ontological facts. If `assert` is used with an ontological statement as in `assert(vin:'Wine'(olp:x))`, then the `assert_into_ontology` predicate is used by the interpreter to add this statement to the semantic knowledge base. That is, the semantic knowledge base is modified by declaring `olp:x` as an instance of the `Wine` concept. On the other hand, if `assert` is used with non-ontological predicates as in `assert(served(vin:'TaylorPort'))`, a new fact is added to the underlying Prolog knowledge base. It should be noted that the addition of a new statement to the semantic knowledge base may make it inconsistent. For example, addition of the statement `rdf:subConceptOf(vin:'Wine', food:'Fruit')` results in an inconsistent semantic knowledge base, because `Wine` and `Fruit` concepts in the wine ontology are defined as disjoint concepts. Therefore, before adding the statement, `assert_into_ontology` checks whether the addition would result in an inconsistency. If the addition would result in an inconsistency, `assert_into_ontology` returns `false` without adding the statement. Otherwise, it modifies the knowledge base and returns `true`. The `retract` predicate works in a similar way: ontological facts are removed from the underlying semantic knowledge base using the `retract_from_ontology` predicate, while others are removed directly from the Prolog knowledge base.

- **Addition and removal of individuals.** New individuals can be created using the `create_individual` predicate. For example, `create_individual(vin:'SoftWine')` creates the individual `SoftWine` within the name-space `vin` as an instance of `owl:Thing`. Then using `assert(vin:'Wine'(vin:'SoftWine'))`, we can declare that `vin:'SoftWine'` is a wine. On the other hand, using the `remove_individual` predicate, we can remove an individual and all statements about that individual from the semantic knowledge base (e.g., `remove_individual(vin:'SoftWine')`).

- **Addition and removal of concepts.** Through the `create_concept` predicate, a new OWL-DL concept can be created based on a DL class description. If the described concept is not satisfiable, the predicate returns `false` without creating the concept; otherwise it returns `true` after creating the concept. A concept description is an OWL-DL class expression [24], which can be a single concept name, a restriction
Table 1. Simple concept description examples.

<table>
<thead>
<tr>
<th>Concept Description</th>
<th>Satisfiable</th>
</tr>
</thead>
<tbody>
<tr>
<td>vin:’Wine’</td>
<td>yes</td>
</tr>
<tr>
<td>(vin:’Wine’; food:’Fruit’)</td>
<td>yes</td>
</tr>
<tr>
<td>(vin:’Wine’, food:’Fruit’)</td>
<td>no</td>
</tr>
<tr>
<td>enum(vin:’TaylorPort’, food:’ThompsonSeedless’)</td>
<td>yes</td>
</tr>
<tr>
<td>(vin:’Wine’, not(enum(vin:’TaylorPort’))</td>
<td>yes</td>
</tr>
<tr>
<td>value(vin:’hasFlavor’, vin:’Delicate’)</td>
<td>yes</td>
</tr>
<tr>
<td>all(inverse(vin:’hasFlavor’), vin:’SauvignonBlanc’)</td>
<td>yes</td>
</tr>
<tr>
<td>some(vin:’hasMaker’, vin:’SaucelitoCanyon’)</td>
<td>yes</td>
</tr>
</tbody>
</table>

on properties, or created using the intersection or the union of two class expressions or the complement of a class expression. A restriction on a property can be specified through someValuesFrom, allValuesFrom, minCardinality, maxCardinality and exact cardinality restrictions [24]. The inverse of a property can also be used in a concept description. Moreover, a concept can be described by enumerating all of its instances; such classes are called enumerated classes [24]. Table 1 shows examples of concept descriptions. OLP also allows the removal of concepts from the semantic knowledge base using remove_concept predicate. When this predicate is used, not only the concept but also all statements about the concept are removed from the semantic knowledge base.

3 Case-Studies

In this section, we introduce two problem domains and shows how OLP has been used by software agents to provide an effective solution to them.

3.1 Resource-Task Matchmaking

In different settings, software agents are expected to fulfill some tasks. However, to achieve a task, an agent may need to have certain resources. Consider Intelligence, Surveillance, Target Acquisition and Reconnaissance (ISTAR) tasks\(^4\), which require sensors and sensing resources. To achieve these tasks, agents should first reason about what types of resources should be used and then resources of these types should be allocated by the agent. However, this kind of reasoning is not trivial in ISTAR domain, e.g., because of the interdependencies of the resources.

We show, in Figure 4, a part of the ontology for the ISTAR domain. In the ontology, the Asset concept represents the resources that could be allocated to tasks. The Platform and System concepts are both assets, but systems may be attached to platforms. Sensors are a specialisation of systems. A sensor needs to be mounted on a platform to work properly. On the other hand, not all platforms can mount every type of sensors. For example, to be used, a radar sensor must be mounted on Unmanned Aerial Vehicles (UAVs), however only specific UAVs such Global Hawk can mount this type of sensors.

\(^4\)http://en.wikipedia.org/wiki/ISTAR
A task may require capabilities, which are provided by the assets. In order to achieve a task, we need to deploy specific assets that provide the required capabilities. Capability requirements of a task are divided into two categories: the first concerns operational capabilities provided by the platforms, and the second concerns intelligence capabilities provided by the sensors attached to a platform. Figure 4 shows Road Surveillance task, which has one operational requirement, namely Constant Surveillance, and one intelligence requirement, namely Imagery Intelligence (IMINT). As shown in the figure, an instance of this task is then defined with two more intelligence requirements (Radar Intelligence and Photographical Intelligence) and an additional operational requirement (High Altitude).

We use the term Deployable Configuration to refer a set of assets required to achieve a task. A deployable configuration of a task is composed of a deployable platform and a set of sensors. A deployable platform provides all operational requirements of the task. Similarly, the sensors in the deployable configuration provide all the intelligence requirements of the task. Furthermore, the deployable platform should have an ability to mount these sensors. Therefore, there is a dependency between the platform and the sensors in a deployable configuration.

An agent can use an OLP program shown in Figure 5 to compute deployable configurations for ISTAR tasks. The OLP program is a Prolog program, where concepts and properties from the underlying ontologies are referenced directly. The getConfigurations predicate computes deployable configurations for a specific task. Each sensor must be carried by a deployable platform that provides all of the operational requirements of the task (e.g., constant surveillance). If a sensor cannot be carried by a deployable platform, there is no point in considering deployable configurations with that sensor type. Using this knowledge, a tailored and efficient matchmaker can be employed. This matchmaker first identifies the deployable platforms that meet the requirements of the task. Once many possibilities are narrowed down by determining deployable platforms, the sensor types that provide the intelligence capabilities required by the task are determined incrementally so that those sensors can be mounted on the deployable platforms.

Fig. 4. The ISTAR ontology on the left and a task instance example on the right.
3.2 Team Formation

In some multi-agent settings, an agent may have a task that cannot be achieved by a single agent. Hence, the agent may need to compose a team of agents, which may cooperate to achieve the task. In this section, we present a case-study to show how the proposed techniques can be used by a hospital agent to compose medical teams to operate in emergency settings. For this purpose, we first need to define a domain ontology to describe emergency tasks, their capability requirements, and resources providing these capabilities. Figure 6 shows an example ontology describing a KidneyTransplantation task with its requirements.

In order to transplant a kidney to a patient with renal failure, we have to compose a surgery team immediately. This team should have expertise in surgery, anesthetics, and nephrology in addition to providing scrub assistance. These requirements are met by the capabilities of resources, which are doctors and nurses in this domain. For the sake of simplicity, we assume capabilities of resources are additive and do not depend on the relationships between them. However, in this case-study, we consider some stereotypes that specify whether a surgery team is considered reliable (i.e., not untrustworthy) according to hospital policy. Stereotypes are beliefs about specific social groups or types of individuals. They are an integral part of human social decision making [14, 16, 17]. Burnett et al. have proposed methods for agents to automatically learn stereotypes based on past experiences [6]. We can use DL to describe untrustworthy surgery teams based on stereotypes about medical staff. Table 2 shows some examples of stereotypes and the untrustworthy team descriptions they imply. Based on these descriptions, we can create sub-concepts of the UntrustworthyTeam concept. The third column in the table shows the names of these concepts.
Fig. 6. Kidney transplantation task, its requirements and resources meeting these requirements.

Fig. 7. A matchmaking mechanism that computes reliable (not untrustworthy) medical teams in emergency settings.
Table 2. Some stereotypes about emergency conditions and the resulting untrustworthy medical team descriptions. These stereotypes are defined using DL in the local ontology of the hospital.

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Untrustworthy Team Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Medical staff becomes unreliable after 12 shift hours.</td>
<td>Team ⊓∃ has.(MedStaff ⊓∃ worked.[minExc(12)])</td>
</tr>
<tr>
<td>Doctors with less than 5 years experience are untrustworthy.</td>
<td>Team ⊓∃ has.(Doctor ⊓∃ medExp.[maxExc(5)])</td>
</tr>
<tr>
<td>Doctors younger than 35 years old are untrustworthy.</td>
<td>Team ⊓∃ has.(Doctor ⊓∃ hasAge.[maxExc(35)])</td>
</tr>
<tr>
<td>Nurses with less than 3 years experience are untrustworthy.</td>
<td>Team ⊓∃ has.(Nurse ⊓∃ medExp.[maxExc(3)])</td>
</tr>
<tr>
<td>Doctors with 10 years or less experience should not work with nurses having less than 5 years experience.</td>
<td>Team ⊓∃ has.(Doctor ⊓∃ medExp.[maxInc(10)]) ⊓∃ has.(Nurse ⊓∃ medExp.[maxExc(5)])</td>
</tr>
</tbody>
</table>

The OLP program in Figure 7 is a matchmaking mechanism designed for this case-study. Medical teams for an emergency task are computed by the `getMedicalTeams` predicate, which gets the name of the task as input and returns a set of medical staff (the surgery team) that is suitable for the task. The computation starts with an empty set of staff, then at each iteration a new member of staff is added to the set if this member provides a capability required by the task but not provided by the other members in the set. The addition of a new member to the set may make the corresponding team untrustworthy, because of the stereotypes. Therefore, the algorithm avoids adding a specific member of staff to the set if this addition makes the corresponding team untrustworthy. This is tested using the `untrustworthy` predicate. Given a set of medical staff, this predicate creates a temporary team instance, whose members are the members of the set. Then, it checks whether the resulting team is also an instance of the `UntrustworthyTeam` concept or not. In this way, untrustworthy teams are detected and eliminated.

4 Related Work and Discussion

There are various extensions and combinations of logic programming with other programming paradigms. One such combination is functional logic programming [2] merging features of logic and functional programming, efficiently implemented in languages such as Curry [11], and experiencing a “revival” due to its appeal to Web programming, notably for scripting purposes. Another extension with a potential wide appeal combines logic programming and object-oriented programming [18,20], making object-oriented programming features such as objects and inheritance available to Prolog programs. Prolog interpreters (e.g., SICStus and Ciao) now commonly allow the seamlessly running, from within a Prolog program, of code implemented in languages such as C or Java. Although these cannot be seen as true extensions, they are very convenient to those wanting to combine functionalities implemented in disparate programming languages.

Rules play an important role in capturing and modeling important domain concepts. Therefore, a lot of effort has been made to develop rule languages and engines for

---

5 http://www-ps.informatik.uni-kiel.de/currywiki/
6 A commercial-standard object-oriented logic programming toolkit aptly named Prolog++ can be found at http://www.lpa.co.uk/pppDet.htm
7 http://www.sics.se/isl/sicstuswww/site/index.html
8 http://clip.dia.fi.upm.es/Software/Ciao
reasoning on top of OWL ontologies. For example, SWRL enables Horn-like rules to be combined with an OWL knowledge base [1]. SWRL aims at extending OWL-DL with semantic rules. Although SWRL does not support negation-as-failure, it implicitly supports classical negation through OWL-DL using classes complements.

Jess [12] is a Java-based expert system shell that uses a RETE algorithm [8] for its forward chaining rule reasoning engine. Jess uses a Common LISP (CLISP) type syntax to describe rules and facts. JessTab [7] is a bridge between Protege [19] and Jess. It enables Jess programs to use and manipulate the knowledge from Protege knowledge bases. This is achieved by mapping Protege knowledge bases to Jess assertions. Originally, JessTab was developed to support Protege-Frames. Thus, JessTab includes only a limited support for handling OWL ontologies. For example, it does not support OWL restrictions and class expressions such as someValuesFrom restrictions while mapping OWL ontologies to Jess assertions. In addition to JessTab, there are some other RETE-based rule engines proposed to work with ontologies. Bossam [13] is one of these rule engines. It supports both negation-as-failure and classical negation. It translates OWL documents into built-in list constructs. Then, the reasoning is made based on these constructs using a RETE algorithm. SweetJess [9] is a defeasible reasoning system based on the Jess expert system shell. Although it supports the Situated Courteous Logic Programs extension of RuleML, it is restricted to simple terms (variables and atoms).

There are some other approaches based on Prolog. SweetProlog [15] is a Java-based system for translating rules into Prolog. It translates OWL ontologies and rules expressed in RuleML into a set of facts and rules in Prolog. Then, the reasoning about these facts and rules are made completely in Prolog. This approach uses JIProlog as a rule engine. Hence, it translates a OWL subset into simple Prolog predicates which a JIProlog engine can handle. The main limitation of SweetProlog is its expressive power as it uses Description Logic Programs (DLP) to enable the integration between ontology and rules. DLP is the intersection of DL and Horn logic programs, so it is less expressive than both DL and Horn logic programs. DR-Prolog [5] is a simple rule-based approach to reasoning with incomplete and inconsistent information. It is compatible with RuleML. It is based on the translation of ontological knowledge into Prolog. The system can reason with rules and ontological knowledge written in RDF Schema (RDFS) or OWL. This is achieved through the transformation of the RDFS constructs and many OWL constructs into rules. Note, however, that a number of OWL constructs cannot be captured. SWORIER [22] is a system that uses Prolog to reason about ontologies and rules in order to answer queries. It translates OWL-DL ontologies with rules in SWRL into Prolog using XSLTs (Extensible Stylesheet Language Transformations). Then, query answering is done in Prolog using this translation. It supports only a subset of OWL-DL constructs.

In the approaches described above, ontological knowledge with rules is translated or mapped to Jess or Prolog assertions. On the other hand, OLP keeps ontological knowledge separated from Prolog programs and transparently delegates ontological reasoning to specialised DL reasoners such as Pellet. Hence, OLP allows a software agent to use the full power of Prolog and the existing reasoners without any loss in the ontological knowledge and expressiveness, as briefly demonstrated in the case-studies.
5 Conclusions

In this paper, we have proposed OLP, a novel tool that combines Logic Programming with Ontological Reasoning. OLP allows software agents to transparently use ontological knowledge and reasoning within logic programs. Unlike similar approaches in the literature, OLP delegates interpretation of ontological predicates to an ontology reasoner during the execution of logic programs. Hence, it takes the full advantage of both ontological reasoning and logic programming without any compromise in expressiveness. Using two case-studies, we have demonstrated how OLP can be used as a tool by agents to solve real-life problems in a practical way.

Acknowledgements

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References


59
Cost-Aware Reorganization Service for Multiagent Systems

Juan M. Alberola, Vicente Julian, and Ana Garcia-Fornes

Departament de Sistemes Informàtics i Computació,
Universitat Politécnica de València, Camí de Vera s/n, 46022, València, Spain,
{jalberola,vinglada,agarcia}@dsic.upv.es

Abstract. Reorganization in Multiagent Systems is aimed at providing support to dynamically adapt the structure and the behaviour of organizations. Current reorganization approaches are mainly focused on providing reorganization solutions that take the benefits of the future organization into account but that do not include the impact of the reorganization costs in the process. Therefore, the costs for achieving future instances of an organization cannot be computed until the reorganization process ends. Organization transition provides a paradigm for relating two different instances of the same organization at different moments. In this paper, we provide a Reorganization Facilitator Service that implements a cost-aware reorganization mechanism that is based on organization transitions. This service provides the associated costs for transition from a current organization to a future organization and the sequence of steps required for this transition. The paper also presents two different examples of organization transition in order to illustrate the use of the proposed service.

1 Introduction

Current trends in the Multiagent Systems (MAS) research community require models that are able to define organizations that can be dynamically adapted according to changes in the environment or in the organization specification. Dynamic adaptation involves modifications in the structure and behavior of a MAS, such as adding, removing or substituting components, that are done while the system is running and without bringing it down [4]. The process that changes an organization into a new one is commonly known as reorganization [10].

Most existing approaches for reorganization in MAS define adaptation processes that are due to organizational changes. Some of these approaches propose solutions for reorganization when the changes prevent the organization from satisfying current goals (such as when an agent leaves the organization) [9, 3]. Other approaches focus on reorganization as a process that is triggered by the domain [17]; however, most of current approaches focus on reorganization to achieve better utility [11, 13]. A reorganization process should provide some kind of increase in utility. Nevertheless, this utility should take into account not only the benefits of acquiring the new organization but also the costs of achieving the new organization.
As stated in [12], human organizations may encounter problems when certain changes are required: they often take longer than expected and desired; the cost of managerial time may increase; and there may be resistance from the people involved in the change. Similarly, in MAS, not every agent is able to change its role at the same cost (for example, the cost for an agent to change its role will not be the same if an agent is acting alone or is interacting with other agents). Nor can every new norm be added at the same cost (for example, some norms may affect every agent of the organization and other norms may only affect a few agents).

Current approaches for reorganization do not take into consideration an evaluation of the costs associated to the reorganization process. Therefore, for the next generation of open and dynamic systems, reorganization models that are able to reason about reorganization not only by considering the profits of the new organization but also the cost of changes are necessary.

By taking into account the costs related to the reorganization process, we can use a reorganization approach that not only provides solutions based on the current instance of the organization but that is also focussed on providing solutions for achieving future instances starting from the current one. This approach allows us to evaluate the impact on the costs of the reorganization before it is carried out. In [2], we presented a cost-based reorganization model based on the concept of organization transition [14]. By using this model, we provide an organization transition mechanism that computes the costs associated to the transition from an initial organization to a future one as well as the sequence of steps required to carry out this transition. Furthermore, there are few infrastructures that provide support for reorganization in MAS. In this paper, we describe the implementation of a reorganization service that provides support for this cost-based organization transition model. This service implements the organization transition mechanism and has been integrated into the Magentix Multiagent Platform [6]. The current implementation of the service is focused on reorganization processes that deal with the problem of role reallocation, which has been dealt with in several works [15, 8]. The objective of this problem is to find the best role assignments to agents.

The rest of the paper is organized as follows. In Section 2, we describe the Organization Transition Model. In Section 3, we detail the implementation of the reorganization service. Then, in Section 4 we show two examples that illustrate the use of the service. Finally, in Section 5, we present some concluding remarks.

### 2 Organization Transition Model

The organization transition model is composed of three parts: the definition of organization; the organization transition that relates two instances of the organization at two different moments; and the computation of the cost related to the organization transition.
2.1 Organization

The organization transition model uses an adaptation of the virtual organization definition proposed by Argente et al. [5]. In this definition, an organization at a specific moment \( t \) is defined as a tuple \( O^t = \langle OS^t, OE^t, \phi^t \rangle \).

The Organizational Specification \( OS \) details the set of elements of the organization by means of two dimensions: \( OS = \langle SD, FD \rangle \), where \( SD \) is the Structural Dimension and \( FD \) is the Functional Dimension. The Structural Dimension \( SD \) describes the set of roles \( R \) contained in the organization at a specific moment. The Functional Dimension \( FD = \langle S, provider \rangle \) describes the set of services \( S \) that the organization is offering at a specific moment. Each service is offered by a set of roles by means the relationship \( provider : S \rightarrow 2^R \).

The Organizational Entity \( OE \) describes the population of agents \( A \) at a specific moment. Finally, the Organizational Dynamics \( \phi = \langle plays, provides \rangle \) represents the relationships among the elements of the \( OS \) and \( OE \):

- \( \text{plays} : A \rightarrow 2^R \), relates an agent with the set of roles that it is playing at a specific moment.
- \( \text{provides} : A \rightarrow 2^S \), relates an agent with the set of services that it is providing at a specific moment.

In order for an agent \( a \) to be able to play the role \( r \), \( a \) must provide all the services \( s \) that \( r \) offers:

\[ \forall \text{plays}(a, r) \in \phi^t \mid \text{provider}(s, r) \in OS^t \rightarrow \text{provides}(a, s) \in \phi^t \]

Therefore, an organization at a specific moment is composed of elements that can be grouped in objects (roles, agents, and agents) and relationships (\( \text{plays} \), \( \text{provides} \), and \( \text{provider} \)). These elements can change during the life-span of the organization.

2.2 Organization Transition

The concept of organization transition was firstly introduced by Matson and De-Louch in [14]. For us, an organization transition allows us to relate two different instances of the same organization at two different moments, initial (\( \text{ini} \)) and final (\( \text{fin} \)). From now on, we refer to two different instances of the same organization as two different organizations. An organization transition function defines how the organization can transition from one organization to another. This transition is carried out by a mechanism that changes the current \( OS^{\text{ini}}, OE^{\text{ini}}, \) and \( \phi^{\text{ini}} \) into a new \( OS^{\text{fin}}, OE^{\text{fin}}, \) and \( \phi^{\text{fin}} \), respectively.

An event \( \varepsilon \) defines each individual change that can be applied to an object or to a relationship during the organization transition in terms of addition or deletion of individual objects or relationships. As an example, we can define an event for addition and deletion such as \( \varepsilon = \text{add}(\text{role}(r)) \) to represent that the role \( r \) has been added to \( R \).
Given two organizations, $O^\text{ini}$ and $O^\text{fin}$, a transition function defines a set of events $\tau$ that allows a transition to $O^\text{fin}$ when applied to $O^\text{ini}$:

$$O^\text{ini} \times \tau \rightarrow O^\text{fin}$$

where $\tau = \{\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n\}$.

An event $\varepsilon$ is dependent of another event $\varepsilon'$ if, in order for $\varepsilon$ to be applied, $\varepsilon'$ must first be applied. The operation $\delta(\varepsilon)$ defines the set of events that $\varepsilon$ is dependent on. The dependency between events defines which ones could be applied simultaneously during the transition process and which ones must be applied sequentially.

A set of events $\tau$ must be split into subsets of events that group independent events. Therefore, a set of events $\tau$ can be represented as a sequence of subsets of events $\tau_1, \tau_2, \ldots, \tau_n$ ordered by a dependency order, i.e., at least one event of $\tau_i$ must be applied before one event of $\tau_{i+1}$, whatever $i$ is.

If a sequence of subsets $\tau_1, \tau_2, \ldots, \tau_n$ is applied to transition from $O^\text{ini}$ to $O^\text{fin}$, the application of each $\tau_i \subset \tau$ causes a transition to an intermediate organization. The sequence of organizations that is reached in the transition between $O^\text{ini}$ and $O^\text{fin}$ represents a transition path between the two organizations.

**Organization transition costs** The application of a set of events provides us with information regarding what changes are required to be carried out in order to fulfill the transition. Thus, by taking this into account, we can associate a transition cost by computing the cost of applying this set of events.

Each event $\varepsilon$ has an associated cost $c(\varepsilon)$ to be applied. Therefore, for any set of events $\tau$ that allow a transition from $O^\text{ini}$ to $O^\text{fin}$, we define the cost of the transition process as the cost of applying all the required events:

$$C_{\text{transition}} = c(\tau) = \sum_{\varepsilon \in \tau} c(\varepsilon)$$

The Organizational Dynamics $\phi^{fin}$ represents relationships between $OS^{fin}$ and $OE^{fin}$. These relationships define which services each agent provides and which roles the agent plays at a specific moment. The problem of role reallocation is to compute the cost of all the events $\tau_\phi$ that allow an Organizational Dynamics transition from $\phi^{ini}$ to $\phi^{fin}$. These events contain plays and provides relationships. This cost defines how costly it is for agents to do the following: to acquire the services to play a specific role, to start playing this role, to stop playing a role that is currently being played by an agent, and to stop providing the services required for this last role. Each one of the possible role reallocations defines a different $\phi^{fin}$ that fulfills $OS^{fin}$ and $OE^{fin}$ and has an associated Organizational Dynamics transition cost of $C_\phi$.

Let $\Theta$ denotes the set of all the possible $\tau_\phi$ that allow an Organizational Dynamics transition from $\phi^{ini}$ and fulfill $OS^{fin}$ and $OE^{fin}$. Our major challenge is to find the specific set of events that minimizes the Organizational Dynamics transition cost: $\tau_{\phi_{\text{min}}} = \arg \min \{\sum_{\varepsilon \in \tau_\phi} c(\varepsilon) \mid \tau_\phi \in \Theta\}$
The transition path of the minimal cost defines a transition from $O^{ini}$ to $O^{fin}$ in which the Organizational Dynamics transition from $\phi^{ini}$ to $\phi^{fin}$ has the associated set of events of the minimal cost $C_{\phi_{min}} = c(\tau_{\phi_{min}})$.

2.3 Cost Computation Organizational Dynamics

The cost of an agent $a$ for playing a role $r$ can be defined as:

$$C_{ACQUIRE}(a, r) = C_{ADD\_SERVICES}(a, r) + C(\text{add}(\text{plays}(a, r)))$$

where $C_{ADD\_SERVICES}(a, r)$ defines the cost of acquiring all the services offered by $r$ that are not already provided by agent $a$:

$$C_{ADD\_SERVICES}(a, r) = \sum C(\text{add}(\text{provides}(a, s)))$$

Then, when agent $a$ provides the services offered by the role $r$, it can acquire the role $r$ for a cost of $C(\text{add}(\text{plays}(a, r)))$.

On the other hand, the cost of agent $a$ to stop playing a role $r$ is defined as:

$$C_{LEAVE}(a, r) = C(\text{delete}(\text{plays}(a, r))) + C_{DEL\_SERVICES}(a, r)$$

where $C(\text{delete}(\text{plays}(a, r)))$ represents the cost of agent $a$ to stop playing the role $r$, and $C_{DEL\_SERVICES}(a, r)$ defines the cost to stop providing all the services offered by $r$ that are no longer required by $a$ for playing other roles:

$$C_{DEL\_SERVICES}(a, r) = \sum C(\text{delete}(\text{provides}(a, s)))$$

Therefore, we can define the cost of role reallocation for agent $a$ from role $r_{old}$ to role $r_{new}$ by taking into account the costs related to stop playing the role $r_{old}$ in order to play the role $r_{new}$:

$$C_{Realloc.}(a, r_{old}, r_{new}) = C_{ACQUIRE}(a, r_{new}) + C_{LEAVE}(a, r_{old})$$

According to this, the cost related to the Organizational Dynamics transition can be now computed as the aggregated cost of each role reallocation:

$$C_{\phi} = \sum_{a \in A} C_{Realloc.}(a, r_{old}, r_{new})$$

where

$$\text{plays}(a, r_{old}) \in \phi^{ini} \land \text{plays}(a, r_{new}) \in \phi^{fin}$$

3 Reorganization Facilitator Service

The Reorganization Facilitator Service uses the Organization Transition Model described in Section 2, and implements an Organization Transition Mechanism that provides support for computing how costly it is for an organization to transition to another organization as well as the sequence of steps required for this transition. This service has been integrated in the Magentix Multiagent Platform.
3.1 Magentix Multiagent Platform

Magentix [6] supports and enables the development and execution of open MAS. It focuses on providing support at the interaction and organization levels, which are key levels in open environments, where heterogeneous agents interact and organize themselves into organizations. Magentix also incorporates modules to provide a tracing service and security support.

Magentix uses the Advanced Message Queue Protocol (AMQP) standard [1] as a foundation for its communication infrastructure. It allows heterogeneous agents to interact with each other via messages that are exchanged using this standard. Magentix provides support to agent organizations by means of the THOMAS architecture [16]. The organization model defined in Section 2.1 is supported by the THOMAS architecture by means of flexible services that can be used by agents:

- **Service Facilitator (SF)**, which allows the registration and search of services provided by internal or external entities by following Service-Oriented Architectures guidelines.
- **Organization Management System (OMS)**, which is in charge of the management of the organizations, taking control of their underlying structure, the roles played by the agents, and their relationships.

The SF and the OMS provide services for managing the life-cycle of the organizations as well as the services provided by the agents. Therefore, systems can be developed where agents are able to dynamically enter and leave the system, change their services, or change the roles that they play in the organizations.

3.2 Reorganization Facilitator Service

The Reorganization Facilitator (RF) service has been implemented as a new module of Magentix (Figure 1). This service provides the support for computing the transition with the lowest cost from an initial organization to a future one.

In order to carry out the organization transition, the agent specify the costs that correspond to events. By using these costs, the RF can be requested to calculate an transition. This makes the RF interact with the OMS to retrieve information regarding the organization that is to be to transitioned from. The RF finds the organization whose transition cost is the lowest and also determines the sequence of steps required to achieve it. Then, the agent can ask the OMS and the SF services to carry out this organization transition (Figure 2).

The RF manages the costs defined to the different events related to an organization transition. The current implementation of the RF provides support for the role reallocation problem. Thus, the events which are considered in the cost computation of an organization transition are those involved in this problem: the addition and deletion of provides and plays relationships. The RF provides the following service to define these costs:

register_transition_costs(OrgID ?OID, CostSpec ?spec)
The $\textit{?OID}$ parameter represents the identifier of the current organization, and the $\textit{?spec}$ parameter represents the specification of the costs.

The $\textit{RF}$ also provides services for assessing the cost of an individual relationship:

\begin{verbatim}
register_add_provides_cost(AgentID ?AID, ServID ?SID, Cost ?cost)
register_delete_provides_cost(AgentID ?AID, ServID ?SID, Cost ?cost)
register_add_plays_cost(AgentID ?AID, RoleID ?SID, Cost ?cost)
register_delete_plays_cost(AgentID ?AID, RoleID ?SID, Cost ?cost)
\end{verbatim}

Depending on the specific service, the $\textit{?SID}$, $\textit{?RID}$, and $\textit{?AID}$ parameters are the service, role, and agent, respectively, for the specific event. The $\textit{?cost}$ parameter defines the cost of this event. As an example, we can define $c$ as the cost of agent $a$ providing the service $s$ using the following request to the $\textit{RF}$:

\begin{verbatim}
register_add_provides_cost(a,s,c)
\end{verbatim}

Once these costs are specified, the $\textit{RF}$ can be asked to calculate an organization transition using the following service:

\begin{verbatim}
request_organization_transition(OrgID ?OID, OrgSpec ?spec)
\end{verbatim}

The $\textit{?OID}$ parameter defines the identifier of the organization to be transitioned from, and the $\textit{?spec}$ parameter is the specification of the future organization to be transitioned to. The tool EMFGormas [7] provides support for specifying this organization.
The RF requests the OMS for the information regarding the current organization. Then the RF is able to calculate the sequence of events that causes a transition to the future organization with the lowest cost. In order to carry out this operation, the RF implements the Organization Transition Mechanism (Figure 3), which is composed of the following three modules:

- The Role realloc org dynamics module calculates the Organizational Dynamics $\phi^{fin}$, which minimizes the organizational transition cost $C_{\phi} = c(\tau_{\phi_{min}})$ from $O_{ini}$ to $O^{fin}$. This module finds the role reallocation with the lowest cost according to both the Organizational Specification and the Organizational Entity of the final organization $OS^{fin}$ and $OE^{fin}$, respectively. Reorganization that involves other components of the organization could be included in future implementations as separate modules.
- Once the organizational dynamics have been calculated, the $O^{fin}$ definition is complete. Thus, the Select events module is in charged of obtaining the set of events required to transition from $O_{ini}$ to $O^{fin}$.
- The Transition path module takes the set of events $\tau$ and calculates the dependency of events. In this case, dependent events must be split into different subsets, providing a sequence that must be applied in order of dependence by defining the transition path between $O_{ini}$ and $O^{fin}$.
- Finally, the Spec generator module uses this sequence of events and generates a sequence of service requests to the OMS and SF. These service requests are returned to the agent along with the cost associated to the organization transition. These requests should be carried out sequentially in order to transition to the future organization. The internal implementation of these modules is out of the scope of this paper.
4 Example

In this section, we present two examples that use the RF service. They are based on an application of tourist services. This application is composed of agents that are grouped into three different organizations: user agents, broker agents, and provider agents. User agents require tourist services and request information regarding the booking of hotels, flights, trains, etc. These agents interact with broker agents in order to obtain the information required. Broker agents maintain tourist information by acting as intermediaries between user agents and provider agents. Provider agents are the agents that belong to the specific hotels, airlines and train companies, etc. As an example of an organization transition, we focus on the organization of broker agents.

By using the organization transition model, we define the organization $O_{ini} = (OS_{ini}, OE_{ini}, \phi_{ini})$ as the current instance of the organization of broker agents. The Organizational Specification $OS_{ini} = (SD_{ini}, FD_{ini})$ defines the Structural Dimension $SD_{ini}$ that specifies the set of roles $R_{ini} = \{r_1, r_2\}$ of the organization at the moment $ini$:

- The role $r_1$ represents the role of Book service provider. Agents that play this role are not able to store information about tourist services, but they are able to interact with agents that store this information (agents that play the role $r_2$).
- The role $r_2$ represents the role of Search service provider. Agents that play this role store information about tourist services and are only capable of interacting with agents that play the role $r_1$.

The Functional Dimension $FD_{ini}$ specifies the set of services $S_{ini}$ offered by the organization at the moment $ini$:

$S_{ini} = \{s_1, s_2, s_3, s_4\}$, where:

- $s_1$ represents the Search hotel service, which provides information about hotels such as availability, prices, suites, location, etc.
- $s_2$ represents the Book hotel service, which provides the service of booking a specific hotel according to specific parameters such as the check-in date, number of nights, breakfast service, etc.
s\textsubscript{3} represents the Search\textsubscript{fly} service, which provides information about flights such as availability, prices, departures, arrivals, etc.

s\textsubscript{4} represents the Book\textsubscript{fly} service, which provides the service of booking a specific flight according to specific parameters such as the city of departure, date, number of unbooked seats, etc.

The \( FD\textsuperscript{ini} \) also specifies the roles that offer each service as the following provider relationships:

\[
\text{provider}(s\textsubscript{2}, r\textsubscript{1}), \text{provider}(s\textsubscript{4}, r\textsubscript{1}), \text{provider}(s\textsubscript{1}, r\textsubscript{2}), \text{provider}(s\textsubscript{3}, r\textsubscript{2})
\]

The Organizational Entity \( OE\textsuperscript{ini} \) defines the current population of agents \( A\textsuperscript{ini} = \{a\textsubscript{1}, a\textsubscript{2}, a\textsubscript{3}, a\textsubscript{4}, a\textsubscript{5}\} \).

Finally, the Organizational Dynamics \( \phi\textsuperscript{ini} \) specifies the services provided by each agent as provides relationships:

\[
\text{provides}(a\textsubscript{1}, s\textsubscript{1}), \text{provides}(a\textsubscript{1}, s\textsubscript{3}), \text{provides}(a\textsubscript{2}, s\textsubscript{1}), \text{provides}(a\textsubscript{2}, s\textsubscript{3}), \\
\text{provides}(a\textsubscript{3}, s\textsubscript{1}), \text{provides}(a\textsubscript{3}, s\textsubscript{3}), \text{provides}(a\textsubscript{4}, s\textsubscript{1}), \text{provides}(a\textsubscript{4}, s\textsubscript{3}), \\
\text{provides}(a\textsubscript{5}, s\textsubscript{2}), \text{provides}(a\textsubscript{5}, s\textsubscript{4})
\]

The \( \phi\textsuperscript{ini} \) also specifies the roles played by each agent as plays relationships:

\[
\text{plays}(a\textsubscript{5}, r\textsubscript{1}), \text{plays}(a\textsubscript{1}, r\textsubscript{2}), \text{plays}(a\textsubscript{2}, r\textsubscript{2}), \text{plays}(a\textsubscript{3}, r\textsubscript{2}), \text{plays}(a\textsubscript{4}, r\textsubscript{2})
\]

Table 1. Number of tourist services provided by each broker agent

<table>
<thead>
<tr>
<th></th>
<th>( a\textsubscript{1} )</th>
<th>( a\textsubscript{2} )</th>
<th>( a\textsubscript{3} )</th>
<th>( a\textsubscript{4} )</th>
<th>( a\textsubscript{5} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hotels</td>
<td>78</td>
<td>112</td>
<td>90</td>
<td>125</td>
<td>0</td>
</tr>
<tr>
<td>Flights</td>
<td>59</td>
<td>95</td>
<td>90</td>
<td>129</td>
<td>0</td>
</tr>
</tbody>
</table>

In this example, a single agent can only play a single role at a given moment. Each broker agent stores information about different agent providers of hotels and flights. This information is provided as tourist services. The number of tourist services provided by each agent can be viewed in Table 1.

According to this information, the current organization \( O\textsuperscript{ini} \) provides information regarding 405 agent providers of hotels and 373 agent providers of flights, as the sum of all the tourist services provided by the agents of the organization. This organization is capable of providing this information to user agents. Therefore, we use these values to represent the utility of the organization at the moment \( ini \), defined in terms of tourist services that are provided at that moment: \( U(O\textsuperscript{ini}) = U(O\textsuperscript{ini}\text{hotels}) + U(O\textsuperscript{ini}\text{flights}) \).
4.1 First Organization Transition

By taking into account the current organization $O^{ini}$, the organization detects that a transition of the current organization is required. In the future organization $O^{fin}$, roles are wanted to be more specialized so that they only offer information about hotels or flights but not both. In this regard, the role $r_3$ will be divided into two different roles, each of which specializes in either hotels or flights. Thus, the future organization $O^{fin}$ is similar to the current one but differs in the set of roles that agents can play as well as in the services offered by each role. To simplify notation, we write $r^{ini}_2$ to refer to the $r_2$ role defined in $O^{ini}$ and $r^{fin}_2$ to refer to the $r_2$ role defined in $O^{fin}$.

The following set of roles $R^{fin} = \{r^{fin}_1, r^{fin}_2, r^{fin}_3\}$ is defined in $O^{fin}$, where:

- The role $r^{fin}_1$ represents the role of Book\_service\_provider. This role offers services regarding both book hotel and flight booking.
- The role $r^{fin}_2$ represents the role of Search\_hotel\_provider. This role offers services regarding the search for hotels.
- The role $r^{fin}_3$ represents the role of Search\_flight\_provider. This role offers services regarding the search for flights.

Furthermore, the roles that offer each service are represented by the following provider relationships:

\[
\text{provider}(s_2, r^{fin}_1), \text{provider}(s_4, r^{fin}_1), \text{provider}(s_1, r^{fin}_2), \text{provider}(s_3, r^{fin}_3)\]

As can be observed, with this more specialized configuration, only a single service is required to play the role $r^{fin}_2$ and the role $r^{fin}_3$.

Specifically, for the future organization $O^{fin}$, a single agent is required for playing the role $r^{fin}_1$, two agents for playing the role $r^{fin}_2$, and two agents for playing the role $r^{fin}_3$. In order to decide which agents are the best candidates for each role in $O^{fin}$, we define the concept of transition cost related to organization utility. As in $O^{ini}$, we define the utility of the future organization $U(O^{fin})$ as the amount of tourist services provided by the agents in $O^{fin}$. Agents that play the role $r^{ini}_1$ are not able to provide information about any tourist service, while agents that play the roles $r^{fin}_2$ and $r^{fin}_3$ are able to provide information about hotels or flights, respectively. By focusing on the utility concept, we define the transition cost for each agent $a$ as the negative impact on the organization utility when $a$ is reallocated to each role of $R^{fin}$.

For the role reallocation problem, the costs related to the plays and provides relationships are specified according to the utility of each agent playing each role. In this example, we consider the costs associated to the plays relationships to be zero, and we only focus on the provides relationships. As an example, $a_1$ provides the $s_1$ and $s_3$ services in $O^{ini}$. Depending on the role that $a_1$ plays in $O^{fin}$, $a_1$ could be required to stop providing them. We represent these costs as:

\[
C(\text{delete}(\text{provides}(a_1, s_1))) = 78
\]
\[
C(\text{delete}(\text{provides}(a_1, s_3))) = 59
\]
As stated in Section 3.2, the costs associated to the relationships involved in the role reallocation problem are sent to the RF using the register_transition_costs service.

After requesting the request_organization_transition service, the RF obtains the role reallocation costs for each agent by means of the Rolerealloc_org_dynamics module (Table 2). As an example, if agent \( a_1 \) is reallocated to the role \( r_{1}^{\text{fin}} \), it must not acquire any service, and it must stop providing the \( s_1 \) and \( s_3 \) services. Thus, its reallocation cost would be:

\[
C_{\text{Realloc}}(a_1, r_2^{\text{ini}}, r_1^{\text{fin}}) = C_{\text{ACQUIRE}}(a_1, r_1^{\text{fin}}) + C_{\text{LEAVE}}(a_1, r_2^{\text{ini}})
\]

\[
C_{\text{ACQUIRE}}(a_1, r_2^{\text{fin}}) = 0
\]

\[
C_{\text{LEAVE}}(a_1, r_1^{\text{ini}}) = C(\text{delete(provides}(a_1, s_1))) + C(\text{delete(provides}(a_1, s_3))) = 78 + 59 = 137
\]

Similarly, if \( a_1 \) is reallocated to the role \( r_2^{\text{fin}} \), it must not acquire any service, and it must stop providing the \( s_3 \) service. If \( a_1 \) is reallocated to the role \( r_3^{\text{fin}} \), it must not acquire any service, and it must stop providing the \( s_1 \) service.

For this organization transition, we assumed that the agents cannot provide new services; therefore, agent \( a_5 \) has an \( \infty \) reallocation cost for the roles \( r_2^{\text{fin}} \) and \( r_3^{\text{fin}} \) because it cannot provide the services \( s_1 \) or \( s_3 \).

Table 2. Role transition cost for each agent

<table>
<thead>
<tr>
<th></th>
<th>( a_1 )</th>
<th>( a_2 )</th>
<th>( a_3 )</th>
<th>( a_4 )</th>
<th>( a_5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1^{\text{fin}} )</td>
<td>137</td>
<td>207</td>
<td>180</td>
<td>254</td>
<td>0</td>
</tr>
<tr>
<td>( r_2^{\text{fin}} )</td>
<td>59</td>
<td>95</td>
<td>90</td>
<td>125</td>
<td>( \infty )</td>
</tr>
<tr>
<td>( r_3^{\text{fin}} )</td>
<td>78</td>
<td>112</td>
<td>90</td>
<td>129</td>
<td>( \infty )</td>
</tr>
</tbody>
</table>

According to these reallocation costs, the Rolerealloc_org_dynamics module calculates the role reallocation that minimizes the transition cost from \( O_1^{\text{ini}} \) to \( O_2^{\text{fin}} \). The Organizational Dynamics \( \phi^{\text{fin}} \) obtained requires an organization transition cost of 373, which is composed of the following plays and provides relationships:

\[
\begin{align*}
\text{plays}(a_5, r_1), \text{plays}(a_1, r_2), \text{plays}(a_2, r_2), \text{plays}(a_3, r_3), \text{plays}(a_4, r_3) \\
\text{provides}(a_1, s_1), \text{provides}(a_2, s_1), \text{provides}(a_3, s_3), \text{provides}(a_4, s_3), \\
\text{provides}(a_5, s_2), \text{provides}(a_5, s_4)
\end{align*}
\]

For this reallocation, the Set of events and Transition_path modules are in charge of providing the sequence of events that causes the organization transition (Figure 4). Then, the Spec_generator module translates this sequence of events into an specification of requests to the SF and the OMS services (Figure 5). This
DeregisterAgentRole(agent1,role2_i)
DeregisterAgentRole(agent2,role2_i)
DeregisterAgentRole(agent3,role2_i)
DeregisterAgentRole(agent4,role2_i)
RemoveProvider(agent1,service3)
RegisterAgentRole(agent1,role2_f)
RemoveProvider(agent2,service3)
RegisterAgentRole(agent2,role2_f)
RemoveProvider(agent3,service1)
RegisterAgentRole(agent3,role3_f)
RemoveProvider(agent4,service1)
RegisterAgentRole(agent4,role3_f)

Fig. 4. Sequence of events

Fig. 5. Service requests specification

specification is returned to the agent which invoked the service. According to the reasoning system of the agent if the organization transition is finally wanted to be carried out, the agent should be made sequentially these requests in order to cause a transition from $O_{ini}^m$ to $O_{fin}^m$.

4.2 Second Organization Transition

In this second organization transition, we explore the impact on the transition cost by a transition that penalizes the agents that do not offer a minimal number of tourist services. This restriction imposes the condition that an agent must provide a minimal number of tourist services in order to play a role.

In the future organization $O_{fin}^m$, each agent must provide a minimal number of hotels to be reallocated to the role $r_{fin}^2$, and a minimal number of flights to be reallocated to the role $r_{fin}^3$. The number of minimal tourist services has been established as 100 for each role. According to this condition, the cost for an agent $a$ to play a role $r$ in which the agent does not provide the minimal number of services offered by this role is $C(\text{add}(\text{plays}(a, r))) = \infty$. As an example, if agent $a_1$ is reallocated to role $r_{fin}^2$, the cost related to the $\text{plays}$ relationship is $\infty$ since $a_1$ does not provide the minimal number of hotels:

$$C_{\text{ACQUIRE}}(a_1, r_{fin}^2) = \infty$$

$$C_{\text{LEAVE}}(a_1, r_{ini}^2) = C(\text{delete}(\text{provides}(a_1, s_3))) = 59$$

$$C_{\text{Realloc.}}(a_1, r_{ini}^2, r_{fin}^2) = C_{\text{ACQUIRE}}(a_1, r_{fin}^2) + C_{\text{LEAVE}}(a_1, r_{ini}^2) = q\infty$$

Similarly to the previous organization transition, the costs are specified, and the $RF$ calculates the role reallocation cost for each agent (Table 3).

However, in this organization transition, there is not a role reallocation that makes it possible to reallocate one agent to the role $r_{fin}^1$, two agents to the role...
Table 3. Role transition cost for each agent

<table>
<thead>
<tr>
<th></th>
<th>$a_1$</th>
<th>$a_2$</th>
<th>$a_3$</th>
<th>$a_4$</th>
<th>$a_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1^{fin}$</td>
<td>137</td>
<td>207</td>
<td>180</td>
<td>254</td>
<td>0</td>
</tr>
<tr>
<td>$r_2^{fin}$</td>
<td>$\infty$</td>
<td>95</td>
<td>$\infty$</td>
<td>125</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$r_3^{fin}$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>129</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

$r_2^{fin}$, and two agents to the role $r_3^{fin}$. Thus, when the RF is requested for a calculation of the organization transition, it obtains an $\infty$ cost. This means that the transition from the current organization $O^{ini}$ to the future organization $O^{fin}$ in this example is not possible. To obtain a valid transition, some condition should be relaxed, external agents should be included, or the possibility of acquiring services should be considered. The RF informs the agent which requested the service that the organization is unachievable from $O^{ini}$.

5 Conclusions

Current approaches for reorganization in MAS are mainly focused on obtaining those reorganizations that provide better utility. However, the computation of the costs related to the reorganization process has not been considered. As in human societies, the costs related to a reorganization should also be taken into account in order to evaluate the result of the reorganization.

With this goal in mind, the organization transition model presented in [2] is focused on evaluating the negative impact of reorganization by means of transition costs. By taking into account these transition costs, we are able to transition from a current organization to a future one. The Reorganization Facilitator presented in this paper provides support for this organization transition model. The service obtains the transition from an initial organization to a future one with the lowest cost. In addition, the sequence of steps required for this transition are provided. This service has been implemented in order to provide this support in a transparent way for the agent that requests the service. The RF interacts with the OMS service to obtain the information required and returns the sequence of steps as requests that should be made to the OMS and SF services.

So far, few infrastructures support the execution of reorganization models. This Reorganization Facilitator Service provides support for reorganization issues. The current Organization Transition Mechanism implementation only considers costs for the role reallocation problem. As future work, we plan to extend this support to specify the cost to other components of the organization.

Acknowledgments

This work has been partially supported by the projects TIN2008-04446 and TIN2009-13839-C03-01. Juan M. Alberola has received a grant from Ministerio de Ciencia e Innovación de España (AP2007-00289).
References

Enforcing Norms in Open MAS

N. Criado¹, E. Argente¹, P. Noriega², and V. Botti¹

¹ Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n. 46022 Valencia (Spain)
Email: {ncriado,eargente,vbotti}@dsic.upv.es
² Institut d’Investigació en Intel.ligència Artificial
Consejo Superior de Investigaciones Científicas
Campus de la UAB, Bellaterra, Catalonia (Spain)
Email: pablo@iiia.csic.es

Abstract. Norms have been promoted as a coordination mechanism for controlling agent behaviours in open MAS. Thus, agent platforms must provide normative support, allowing both norm-aware and non norm-aware agents to take part in MAS controlled by norms. In this paper, the most relevant proposals on the definition of norm enforcement mechanisms have been analysed. These proposals present several drawbacks that make them unsuitable for open MAS. In response to these open problems, this paper describes a new Norm-Enforcing Architecture aimed at controlling open MAS.

1 Introduction

One of the main applications of Multi-Agent Systems (MAS) is its usage for supporting large scale open distributed systems. These systems are characterized by the heterogeneity of their participants; their limited trust; a high uncertainty; and the existence of individual goals that might be in conflict [2]. In these scenarios, norms are conceived as an effective mechanism for achieving coordination and ensuring social order.

This paper points out the main deficiencies and drawbacks of current platforms and infrastructures when supporting norms. With the aim of enforcing norms in open MAS, in this paper a Norm-Enforcing Architecture is proposed. Specifically, our Norm-Enforcing Architecture has been integrated into the Magentix platform³. The Magentix platform allows the management of open MAS in a secure and optimized way. Its main objective is to bring agent technology to real domains: business, industry, e-commerce, among others. This goal entails the development of more robust and efficient mechanisms for enforcing norms that control these complex applications.

This paper is organized as follows: Section 2 contains the analysis of the main proposals on norm enforcement; Section 3 describes briefly the Magentix platform; Section 4 describes the proposed Norm-Enforcing Architecture; and, finally, Section 5 contains a conclusion and future works.

³ http://users.dsic.upv.es/grupos/ia/sma/tools/magentix2/
2 Related Work

In general, norms represent an effective tool for regulating the actions of software agents and the interactions among them. Most of proposals on norms for controlling MAS tackle this issue from a theoretical perspective [5, 24]. However, there are also works on norms from a computational point of view. These approaches are focused on giving a computational interpretation to norms in order to use them in the execution of MAS applications. In this sense, they illustrate how MAS platforms and infrastructures can be extended to implement norms, given that the internal states of agents are not accessible. Therefore, norms cannot be imposed as agent’s beliefs or goals, but they must be implemented in the platforms by means of control mechanisms [17].

These control mechanisms are classified into two categories [17]: regimentation and enforcement mechanisms. Regimentation mechanisms consist on making the violation of norms impossible by mediating the resources and the communication channel, as in case of the Electronic Institutions. However, the regimentation of all actions can be not only difficult or impossible, but also sometimes it is preferable to allow agents to violate norms [7]. In response to this, the enforcement mechanisms are applied after the detection of the violation of a norm, reacting upon it.

Proposals on the enforcement of norms can be classified according to the entities that are in charge of observing norm compliance. There are proposals in which those agents involved by an interaction are responsible for monitoring norms. In these approaches, agents evaluate subjectively to their interaction partners. In accordance with this evaluation, agents may punish or reward their partners [4] or they may start a grievance procedure [9].

If there are agents not directly involved by an interaction that are capable of observing it, then they would be also capable of forming an own image about the interacting participants. Moreover, these evaluations or reputations may be exchanged. Thus, agents are persuaded to obey norms because their non-normative behaviour can be observed by others. In this case, the society as a whole acts as norm enforcer [23]. These non-compliant agents might be even excluded from the society [11]. The role of emotions in the social enforcement [12] is also interesting. For example, the work described in [13] models the emotion-based enforcement of norms in agent societies. In this approach, the whole society observes compliance of norms and generates social emotions such as contempt or disgust, in case of norm violation; and admiration or gratefulness, in case of norm compliant behaviour. In the same way, agents observe the expression of these emotions and are also able to generate emotions such as shame or satisfaction in response. The main drawback of these proposals is the fact that the infrastructure does not offer support for enforcing norms. Thus, the norm monitoring and reaction to violations must be implemented by agent programmers at user level. In this sense, agent programmers are responsible for watching over norm compliance. Even if the infrastructure provides authority entities that act as arbiters or judges in grievance processes, agents must be endowed with capabilities for both detecting of norm violations and participating in these dispute resolution processes.
Usually, agent platforms provide entities that are in charge of both observing and enforcing norms. The work contained in [19] proposes a distributed enforcement mechanism in which each agent has an interface that sends legal messages and enforces obligations. One of the main drawbacks of this proposal is the fact that norms can be only expressed in terms of messages sent or received by a single agent; i.e., this framework does not support the definition of norms that affect an agent as a consequence of a message independently sent by another agent. This problem is solved by Gaertner et al. in [16]. In this work, Gaertner et al. propose a distributed architecture for enforcing norms in EI. Specifically, dialogical actions performed by agents are taken into account by the normative level; i.e., a higher level in which norm reasoning and management processes are performed in a distributed manner. Norms only control the illocutions performed by agents, whereas non-illocutive actions and states of affairs cannot be controlled by this approach. Modgil et al. propose in [20] a general architecture for monitoring norm-governed systems. In particular, this architecture takes an overhearing approach; i.e., all messages exchanged among agents are observed and processed. Thus, it is a two layer architecture in which observers (i.e., the lower layer) are capable of reporting to monitors (i.e., the higher layer) on states of interest relevant to the activation, fulfilment, violation and expiration of norms. In this paper, we also propose a two layer approach to norm enforcement. However, in our approach the reasoning about norm enforcement is performed in the two layers whereas in the proposal of Modgil et al. the reasoning process is performed only by monitors. Moreover, our proposal takes as a reference a trace event system based on a publish/subscription procedure (this trace event system is explained in Section 3.1). It reduces appreciably the number of messages exchanged in the platform for controlling norms. Finally, the proposal of Modgil et al. does not give any detail of how the monitoring and observation levels can be dynamically distributed into a set of coordinated entities in response to a changing environment.

Finally, there are works that use a mixed approach for controlling norms [10, 18]. In this sense, they propose the use of regimentation mechanisms for ensuring compliance with norms that preserve the integrity of the application. Moreover, institutional enforcement is also used for controlling norms that cannot be regimented due to the fact that they are not verifiable or their violation may be desirable. In these two proposals only the norms controlling access to the platform are controlled whereas other problem domain norms are not automatically controlled.

As being illustrated by this section, existing proposals that provide support to norm-enforcing present some drawbacks that make them unsuitable for controlling norms in open MAS. In summary, the most important requirements for norm-enforcing architectures are:

- **Automatic Enforcement.** It must provide support for the detection of norm violations and the application of remedial mechanisms. It implies that agents can trust the enforcement system that will sanction their partners if they behave dishonestly. Moreover, the enforcement architecture must pro-
vide normative information in order to allow norm-aware agents to realise that they or other agents have violated a norm. Thus, agents are persuaded to obey norms not only by a material system of sanctions but also since their non-normative behaviour can be observed by others, which may reject to interact with them in the future.

- **Control of general norms.** It must control complex and general norms. Thus, it must allow the definition and management of norms that control not only the messages exchanged among agents but also other actions carried out by agents. In addition, it must support the enforcement of norms that control states of affairs. Finally, it must bring the possibility of controlling norms that are defined in terms of actions and states of affairs that occur independently (e.g., actions that are performed by different agents).

- **Dynamic.** Dynamic situations may cause norms to lose their validity or to need to be adapted. Thus, norm-enforcing mechanisms should provide solutions to open MAS in which the set of norms evolves along time. Moreover, it must provide support for the enforcement of unforeseen norms that control activities and actions that are defined on-line.

- **Efficient, Distributed and Robust.** Finally, enforcement mechanisms must bring the possibility of performing this task in a distributed way. Therefore, they must be unlikely to fail. Thus, this distributed architecture must be capable of operating quickly and effectively in an organized way.

With the aim of meeting these requirements, we propose in Section 4 a Norm-Enforcing Architecture for controlling norms in the Magentix platform. Thus, the Norm-Enforcing Architecture takes as basis the organization and interaction support offered by Magentix. Next, the Magentix platform is briefly described.

### 3 The Magentix Platform

Magentix is an agent platform for open MAS in which heterogeneous agents interact and organize themselves into Virtual Organizations (VO) [15]. Thus, it provides support at two levels:

- **Organization level.** Magentix provides access to the organizational infrastructure [1] through a set of services included on two main components: the Service Facilitator [25], which is a service manager that registers services provided by entities and facilitates service discovering for potential clients; and the Organization Management System (OMS) [10], which is in charge of VO management, taking control of their underlying structure, the roles played by the agents and the norms that govern the system behaviour.

- **Interaction level.** Magentix provides support to: agent communication, supporting asynchronous reliable message exchanges and facilitating the interoperability between heterogeneous entities; agent conversations [14], which are automated Interaction Protocols; tracing service support [6], which allows agents in a MAS to share information in an indirect way by means of trace events; and, finally, Magentix incorporates a security module [3] that
provides key features regarding security, privacy, openness and interoperability.

Norms define what is considered as permitted, forbidden or obliged in an abstract way. However, norm compliance must be controlled considering the actions and messages exchanged among agents at the interaction level. The Norm-Enforcing Architecture proposed in this paper tries to fill the gap between the organizational level, at which norms are managed by the OMS; and the interaction level, at which actions and communications between agents can be traced. Next, the Tracing Service Support and the management of norms, provided by the OMS, are described in more detail.

3.1 Tracing Service Support

In order to facilitate indirect communication, Magentix provides Tracing Service Support [6]. This service is based on the publish/subscribe software pattern, which allows subscribers to filter events attending to some attributes (content-based filtering), so that agents only receive the information they are interested in and only requested information is transmitted. In addition, security policies define what entities are authorized to receive some specific events. These tracing facilities are provided by a set of components named Trace Manager (TM).

A trace event or event is a piece of data representing an action or situation that has taken place during the execution of an agent or any other component of the MAS. An event $e$ is defined as a tuple $\langle Type, Time, Origin, Data \rangle$ where: $Type$ is a constant that represents the nature of the information represented by the trace event; $Time$ is a numeric constant that indicates the global time at which the event was generated; $Origin$ is a constant that identifies the entity that has generated the event; and $Data$ contains extra attached data required for interpreting the event. Trace events can be processed or even combined in order to generate compound trace events, which can be used to represent more complex information.

There can be three types of tracing entities (i.e., those elements of the system capable of generating and/or receiving events): agents, artefacts or aggregations of agents. Any tracing entity of the system is provided with a mailbox for receiving or delivering events ($E_{In}$ and $E_{Out}$). In this sense, entities that want to receive certain trace events request the subscription to these events to the TM by adding an event template to their subscription template list ($Sub$). An event template is a tuple $\langle Type, Origin, Data \rangle$ where $Type$, $Origin$ and $Data$ are the filtering specified criteria.

3.2 Organization Management System (OMS)

The Organization Management System (OMS) [10] is responsible for the management of VO and their constituent entities. In order to allow this management, the OMS provides a set of services classified in: structural services, which comprise services for adding/deleting norms ($RegisterNorm$ and $DeregisterNorm$ services), adding/deleting roles and groups; informative services, that provide...
information of the current state of the organization; and **dynamic services**, which allow agents to enact/leave roles inside VOs *(AcquireRole and LeaveRole services).* Moreover, agents can be forced to leave a specific role *(Expulse service).* When the OMS provides successfully any of these services, then it generates an event for informing about the changes produced in the VO.

The *RegisterNorm/DeregisterNorm* services allow entities to modify the norms that are in force (i.e., that are applicable) within the VO. In particular, a norm is defined as a conditional rule that defines under which conditions obligation, permission and prohibition instances should be created [22]. A *norm* is defined as a tuple \( n = \text{id} : (D, T, A, E, C, S, R) \) where: \( \text{id} \) is the norm identifier; \( D \in \{F, O\} \) is the deontic modality of the norm, \( F \) represents prohibition and \( O \) represents obligation; \( T \) is the target of the norm, the role to which the norm is addressed; \( A \) the norm activation condition, it defines under which circumstances the norm is active and must be instantiated; \( E \) is the norm expiration condition that determines when the norm expires and no longer affects agents; \( C \) represents the action or state of affairs that is forbidden or obliged; \( S \) and \( R \) describe the sanctioning and rewarding actions that will be carried out in case of norm violation or fulfilment, respectively. This work takes a closed world assumption where everything is considered as permitted by default. Therefore, permissions are not considered in this paper, since they can be defined as normative operators that invalidate the activation of an obligation or prohibition. As previously argued, our Norm-Enforcing Architecture builds on the event tracing approach to monitoring. Thus, all the norm conditions (i.e., \( A, E \) and \( C \)) are expressed in term of events.

Once the activation condition of a norm holds; i.e., the activation event is detected, then it becomes active and several norm *instances* (or instances for short), according to the possible groundings of the activation condition, must be created. Thus, given a perceived event \( e \), a norm \( n = \text{id} : (D, T, A, E, C, S, R) \) is instantiated into an instance \( i = \text{id} : (D, T, E', C', S', R') \) where: there is a substitution \( \sigma \) such as \( e = \sigma(A) \); \( C' = \sigma(C) \); \( E' = \sigma(E) \); \( S' = \sigma(S) \); and \( R' = \sigma(R) \).

From that moment on, a new instance is created and all agents playing the target role are under its influence. Thus, a normative *power* (or power for short) represents the control over a concrete agent that is playing the target role of an instance. Thus, a power is defined as a tuple \( p = \text{id} : (D, T, AgentID, C, S, R, W) \) where: \( \text{id}, D, T, C, S, R \) are defined as in case of instances; \( AgentID \) is a constant that identifies the agent affected by the power; and \( W \) is a boolean constant that expresses if the condition \( C \) has been detected or not (i.e., it the event \( C \) has been received).

The next section describes the Norm-Enforcing Architecture proposed in this paper. It is a two layer architecture formed by: a higher level responsible of detecting the instantiation of norms; and a lower level in charge of enforcing powers on agents. Thus, the operational semantics of norms, instances and powers (i.e., how they are created, deleted, fulfilled and violated) is explained in the following section.
4 Norm-Enforcing Architecture

The main purpose of the architecture described in this section is to endow the Magentix platform with a norm enforcing framework that is capable of controlling norms in open applications in which unforeseen scenarios may occur. For this reason, this Norm-Enforcing Architecture has been distributed into two layers. In particular, the higher layer is formed by Norm Manager (NM) entities that control all processes related with the creation and elimination of both norms and instances. The lower layer is formed by Norm Enforcing (NE) entities that are responsible for controlling the agents' behaviours. Next, the NM and NE entities are described in detail.

4.1 Norm Manager

The Norm Manager (NM) is responsible for determining what norms are active (i.e., have been instantiated) in a given moment. Algorithm 1 illustrates pseudocode of the control loop performed by the NM. Each time the NM receives an event \(e\), then it adds the event to the store of received events \(E_{st}\) and handles the event according to the event type. Mainly, the NM carries out a process that can be divided into two differentiated tasks: norm management and instance management. Thus, the MN maintains a list \(N\) that contains all norms that have been registered in Magentix and a list \(I\) that contains all instances that remain active at a given moment.

**Norm Management.** In order to maintain the norm list, the NM subscribes to those events sent by the OMS related to the creation and deletion of norms (i.e., RegisterNorm and DeregisterNorm events). Thus, any time the NM receives an event \(e\), it adds the event to the store of received events \(E_{st}\) and handles the event according to the event type. Mainly, the NM carries out a process that can be divided into two differentiated tasks: norm management and instance management. Thus, the MN maintains a list \(N\) that contains all norms that have been registered in Magentix and a list \(I\) that contains all instances that remain active at a given moment.

When a norm is deregistered, then the NM removes it from its norm list. Moreover, it removes all instances that have been created out of this norm. For each one of these deleted instances, the NM unsubscribes from the expiration event (i.e., it removes the template \(\langle E', \neg, \neg \rangle\) from \(Sub\)) and generates an event for informing about the deletion of the instance (i.e., a NormDeletion event is sent through the event sending box).

**Instance Management.** Once the activation event of a norm is received (i.e., matches\((e, A)\)), then the NM instantiates the norm (i.e., instantiation\((e, n)\)) and adds it to the instance list. At this moment, the NM subscribes to the expiration event and informs about the activation of the norm (i.e., the InstanceActivation event is sent by the NM).

Similarly, when the NM receives the expiration event of any instance (i.e., matches\((e, E)\)), then it removes it from the instance list, unsubscribes from
the expiration event and informs about the expiration of this instance (i.e., the *InstanceExpiration* is sent by the NM).

Initially, there is a single NM registered in the Magentix platform. However, the NM is capable of simple adaptation behaviours (i.e., replication and death) in response to changing situations. For example, before the NM collapses (i.e., its event reception box is full), then it might replicate itself and remove its subscription to the *RegisterNorm* event. Thus, the new NM would be responsible for controlling the activation of the new norms. Similarly, if the NM reaches a state in which it has no norm to control and it is not the last NM subscribed to the *RegisterNorm* event, then it removes itself. These replication and death mechanisms are a simple example that illustrates how the higher layer of the Norm-Enforcing Architecture can be dynamically distributed into several NMs. However, the definition of more elaborated procedures for adapting dynamically to changing environments [21] is a complex issue that is over the scope of this paper.

### 4.2 Norm Enforcer

The Norm Enforcer (NE) is responsible for controlling agent behaviours. Thus, it detects violations and fulfilsments of norms, and reacts upon it by sanctioning or rewarding agents. Algorithm 2 illustrates the control loop executed by the NE. As illustrated by this algorithm, the NE maintains a list (*I*) with the instances that hold in a given moment. Thus, it subscribes to the events sent by the NM that inform about the activation and expiration of instances, and deletion of norms. Besides that, the NE is also in charge of controlling agents affected by the instances. Thus, it maintains a list (*P*) that contains all powers that have been created out of the instances. In order to determine what agents are controlled by these instances, it also maintains a list (*RE*) containing information about role enactment (i.e., the set of roles that each agent is playing at a given moment). In order to update this list, the NE subscribes to the events sent by the OMS that inform about the fact that an agent has acquired or left a role (*AcquireRole* and *LeaveRole* events). In addition, the NE also subscribes to the *Expel* event, which informs about the fact that a particular agent has been forced to leave a role as a disciplinary measure.

As in case of the NM, the NE starts by retrieving an event from its event reception box. Then, different operations are performed according to the type of the event that has been received. In concrete, the NE carries out a process that can be divided into three different activities: role enactment management, instance management and observation of behaviours.

**Role Enactment Management.** Algorithm 3 illustrates pseudocode corresponding to the role enactment management process. Specifically, if the OMS informs that an agent (*AgentID*) has acquired a new role (*RoleID*), then the NE updates the role enactment list. Moreover, the list of instances is also checked.
Algorithm 1 Norm Manager Control Loop

Require: Event reception box $E_{In}$

Require: Event sending box $E_{Out}$

Require: Event store $E_{St}$

Require: Subscription list $Sub$

Require: $(RegisterNorm, OMS, −)$ in $Sub$

Require: $(DeregisterNorm, OMS, −)$ in $Sub$

Require: Norm list $N$

Require: Instance list $I$

1: while $E_{In}$ is not empty do
2:   Retrieve $e$ from head of $E_{In}$ // $e = \langle Type, Time, Origin, Data \rangle$
3:   Add $e$ to $E_{St}$
   // Norm Management
4:   if $Type = RegisterNorm$ then // $Data = id : \langle D, T, A, E, C, S, R \rangle$
5:      Add $Data$ to $N$
6:      Add $\langle A, −, − \rangle$ to $Sub$
7:   end if
8:   if $Type = DeregisterNorm$ and $Data$ in $N$ then // $Data = id : \langle D, T, A, E, C, S, R \rangle$
9:      Remove $Data$ from $N$
10:     Remove $\langle A, −, − \rangle$ from $Sub$
11:    for all $i$ in $I$ do // $i = id' : \langle D', T', E', C', S', R' \rangle$
12:       if $id' = id$ then
13:          Remove $i$ from $I$
14:         Remove $\langle E', −, − \rangle$ from $Sub$
15:        Add $(NormDeletion, NM, id : \langle D', T', E', C', S', R' \rangle)$ to $E_{Out}$
16:     end if
17:    end for
18: end if
19: // Instance Management
20:   for all $n$ in $N$ do // $n = id : \langle D, T, A, E, C, S, R \rangle$
21:      if matches($e$, $A$) then // the norm is active
22:         $i = instantiation(e, n)$ // $i$ is an instance
23:      if $i$ not in $I$ then
24:         Add $i$ to $I$
25:        Add $(InstanceActivation, NM, i)$ to $E_{Out}$
26:      end if
27:   end if
28: end for
29:   for all $i$ in $I$ do // $i = id : \langle D, T, E, C, S, R \rangle$
30:      if matches($e$, $E$) then
31:         Remove $i$ from $I$
32:        Remove $\langle E, −, − \rangle$ from $Sub$
33:       Add $(InstanceExpiration, NM, i)$ to $E_{Out}$
34:    end if
35:   end for
36: end while
Algorithm 2 Norm Enforcer Control Loop

Require: Event reception box $E_{In}$
Require: Event sending box $E_{Out}$
Require: Event store $E_{St}$
Require: Subscription list $Sub$
Require: ⟨NormDeletion, NM, −⟩ in $Sub$
Require: ⟨InstanceActivation, NM, −⟩ in $Sub$
Require: ⟨InstanceExpiration, NM, −⟩ in $Sub$
Require: ⟨AcquireRole, OMS, −⟩ in $Sub$
Require: ⟨LeaveRole, OMS, −⟩ in $Sub$
Require: ⟨Expel, OMS, −⟩ in $Sub$
Require: Instance list $I$
Require: Power list $P$
Require: Role enactment list $RE$

1: while $E_{In}$ is not empty do
2: Retrieve $e$ from $E_{In}$ // $e = \langle Type, Time, Origin, Data \rangle$
3: Add $e$ to $E_{St}$
   ... // Role enactment management
   ... // Instance management
   ... // Observation of Behaviour
66: end while

for determining what of the instances affect the RoleID. For each one of these instances, the NE creates a new power addressed to the agent identified by AgentID. In addition, the NE subscribes to the event expressed in the norm condition in order to be aware of the fulfilment or violation of this norm (i.e., it adds the event template $\langle C, \text{AgentID}, − \rangle$ to the subscription list $Sub$).

On the contrary, if the OMS informs that an agent is not longer playing a role, then the role enactment list is updated. Similarly, all powers that affect the AgentID as a consequence of being playing RoleID are removed. Therefore, the NE does not have to observe the norm condition and unsubscribes from this event. Finally, if any agent leaves a role voluntarily (i.e., the LeaveRole event is received) before fulfilling its pending obligations, then it will be sanctioned (i.e., the NE would perform the sanctioning action $S$). Moreover, the NE would inform about the fact that an agent has been sanctioned for non-compliance with an obligation (i.e., the Sanction event is sent through the $E_{Out}$ box).

Instance Management. This process is contained in Algorithm 4. If the NE is informed by the NM the creation of a new instance (i.e., the InstanceActivation event is received), then the NE updates its instance list and creates new powers for controlling all agents that are playing the target role at this moment. The watch condition ($W$) of the new powers is initially set to false. Thus, for each one of the new powers the NE starts to observe indirectly norm compliance by subscribing to the event $C$. 
Algorithm 3 Role Enactment Management

4: if Type = AcquireRole then // Data is a pair (AgentID, RoleID)
5: Add Data to RE
6: for all i in I do // i = id : (D, T, E, C, S, R)
7: if T = RoleID then
8: Add id : (D, T, AgentID, C, S, R, false) to P
9: Add ⟨C, AgentID, −⟩ to Sub
10: end if
11: end for
12: end if
13: if Type = LeaveRole or Type = Expel then // Data is a pair (AgentID, RoleID)
14: Remove Data from RE
16: if T = RoleID then
17: Remove p from P
18: Remove ⟨C, AgentID, −⟩ from Sub
19: if D = O and W = False and Type = LeaveRole then
20: Perform S// against AgentID
21: Add ⟨Sanction, NE, Violated(id, AgentID)⟩ to EOut
22: end if
23: end if
24: end for
25: end if

If an instance has no longer effect (i.e., the InstanceExpiration or NormDeletion events is perceived), then the NE updates the instance list and removes all powers created out of this instance. An instance becomes ineffective since the expiration condition hold or since the norm that has given rise to it has been abolished. In the first case (i.e., the InstanceExpiration event is received), the agent is responsible for fulfilling the norm. Thus, if the instance obliges an agent to reach some state of affairs (e.g., the agent is obliged to perform an action) and this state has not been observed yet (i.e., the watch condition W is false), then the agent will be sanctioned. On the contrary, if the agent is prohibited to reach some situation and the forbidden state has not been observed (i.e., W is false) then the agent will be rewarded. Finally, if an instance becomes ineffective due to the deletion of a norm, then the agent is not responsible for the fulfilment of the norm and enforcement actions are not performed.

Observation of Behaviours. The observation of behaviours corresponds to pseudocode contained in Algorithm 5. The NE checks for each one of the powers if the C event has been detected (i.e., matches(e, C)). If it is the case, then the power is updated. Specifically, the watch condition is registered as true indicating that the norm condition has been perceived. Next, enforcement actions are performed according to the deontic modality of the power. For example, power is an obligation then the obligation is considered as fulfilled (i.e., the power is
Algorithm 4 Instance Management

26: if $Type = InstanceActivation$ then // Data = id : $\langle D, T, E, C, S, R \rangle$
27:   Add Data to $I$
28: for all $(AgentID, RoleID)$ in $RE$ do
29:   if $RoleID = T$ then
30:     Add id : $\langle D, T, AgentID, C, S, R, false \rangle$ to $P$
31:     Add $\langle C, AgentID, - \rangle$ to $Sub$
32:   end if
33: end for
34: end if
35: if ($Type = InstanceExpiration$ or $Type = NormDeletion$) and Data in $I$ then // Data = id : $\langle D, T, E, C, S, R \rangle$
36:   Delete Data from $I$
37: for all $p$ in $P$ do // $p = id : \langle D, T, AgentID, C, S, R, W \rangle$
38:   Remove $p$ from $P$
39:   Remove $\langle C, AgentID, - \rangle$ from $Sub$
40: if $Type = InstanceExpiration$ then // The agent is responsible for norm fulfilment
41:   if $W = false$ and $D = O$ then // The obligation has not been fulfilled before the deadline
42:     Perform $S$ // against AgentID
43:     Add $\langle Sanction, NE, Violated(id, AgentID) \rangle$ to $E_{Out}$
44:   end if
45:   if $W = false$ and $D = F$ then // The prohibition has not been observed
46:     Perform $R$ // in favour of AgentID
47:     Add $\langle Reward, NE, Fulfilled(id, AgentID) \rangle$ to $E_{Out}$
48:   end if
49: end if
50: end for
51: end if
deleted from \( P \) and the agent is rewarded. Similarly, if it is a prohibition then the agent will be sanctioned.

As in case of the NM, the lower level of the Norm-Enforcing Architecture has been described as it was formed by a single entity. However, this layer may be formed by a set of specialized NEs. For example, the set of instances can be distributed among NEs according to the target role. Thus, each NE is responsible for controlling actions in which a specific set of roles is involved. It is also possible to specialize NE for controlling a specific group of agents independently of the roles that they play. Finally, it is also possible the dynamic adaptation of the amount of NEs by performing cloning and self-deletion operations.

### Algorithm 5 Observation of Behaviours

```plaintext
52: for all \( p \) in \( P \) do // \( p = \text{id} : \langle D, T, \text{AgentID}, C, S, R, W \rangle \)
53:   if \( \text{matches}(e, C) \) then
54:     Remove \( p \) from \( P \)
55:   if \( D = \text{false} \) then // The prohibition has been violated
56:     Add \( \text{id} : \langle D, T, \text{AgentID}, C, S, R, \text{true} \rangle \) to \( P \)
57:     Perform \( S \) against \( \text{AgentID} \)
58:     Add \( \langle \text{Sanction, NE, Violated(id, \text{AgentID})} \rangle \) to \( E_{\text{Out}} \)
59:   else // The obligation has been fulfilled and it expires
60:     Perform \( R \) in favour of \( \text{AgentID} \)
61:     Add \( \langle \text{Reward, NE, Fulfilled(id, \text{AgentID})} \rangle \) to \( E_{\text{Out}} \)
62:     Remove \( \langle C, \text{AgentID}, - \rangle \) from \( \text{Sub} \)
63:   end if
64: end if
65: end for
```

5 Conclusions and Future Works

In this paper, we have described a Norm-Enforcing Architecture considering the facilities provided by the Magentix platform. The main aim of this Norm-Enforcing Architecture is to overcome problems of existing proposals on norm enforcement. Thus, the requirements taken into account by our proposal are:

- **Automatic Enforcement.** Our proposal enforces norms providing support to those agents that are not endowed with normative reasoning capabilities. In addition, the generation of events for informing about sanctions and rewards allows norm-aware agents to use this information for selecting the most suitable interaction partners.

- **Control of general norms.** Our definition of norm is based on notion of event. An event represents an action or situation that has taken place during the execution of any tracing entity (i.e., an agent, aggregation of agents or an...
artefact). Thus, norms are defined in terms of events that can be generated independently by different tracing entities.

- **Dynamic.** Magentix allows the dynamic modification of norms. Accordingly, our proposal has been designed taking into account the possibility that norms can be created or deleted on-line.

- **Efficient, Distributed and Robust.** Finally, our Norm-Enforcing Architecture is build upon a trace event system, which provides support for indirect communication in a more efficient way than overhearing approaches. Besides that, we have provided a preliminary solution to the adaptation of the architecture in response to situations in which the number of agents or norms to be controlled dramatically changes.

As future work, we plan to deal with complex scenarios in which there are norms whose violation cannot be directly observed, since they regulate situations that take place out of the institution boundaries. Or even more, norms can be interpreted ambiguously. This entails the development of intelligent and proactive norm-enforcing entities (i.e., agents) [8] endowed with capabilities for applying techniques such as negotiation or conflict resolution procedures.

6 Acknowledgments

This paper was partially funded by the Spanish government under grants CONSOLIDER-INGENIO 2010 CSD2007-00022, TIN2009-13839-C03-01, TIN2008-06701-C03-03,TIN2008-04446 and by the FPU grant AP-2007-01256 awarded to N. Criado. This research has also been partially funded by the Generalitat de Catalunya under the grant 2009-SGR-1434 and Valencian Prometeo project 2008/051.

References

Evolving Semantics for Agent-based Collaborative Search

Murat Şensoy
Department of Computing Science, University of Aberdeen, AB24 3UE, Aberdeen, UK
m.sensoy@abdn.ac.uk

Abstract. Millions of users search the Web every day to locate Web resources relevant to their interests. Unfortunately, the Web resources found by a user with a specific interest are usually not shared with others having the same or similar interests. In this paper, we propose an agent-based approach for collaborative distributed semantic search of the Web. Our approach enables a human user to semantically describe his search interest to an agent. Depending on the interests of their users, the agents evolve their vocabularies and create search concepts. Based on these search concepts, the agents discover other agents having similar search interests and collaborate with them to locate Web resources relevant to their search interests. Our empirical evaluations and the analysis of the proposed approach show that our approach enables agents with similar interests to coordinate and compose virtual communities. Within these communities, the agents interact to locate and share pointers to the Web resources relevant to the search interests of their users.

1 Introduction

When we need a specific information, we usually have an intuition that it relies somewhere on the Web. At this point, the main challenge becomes locating the Web resources containing the information we seek. However, as the volume of the Web increases and its content becomes more bogus, results of search engines become more confusing. Although the search results are ordered according to the relevance or importance, their number is usually in the order of 1000s. Therefore, it is the task of users to mine piles of the returned results to figure out which resources actually contain the information they seek. Search engines that are using page-rank like algorithms give higher rankings for well-known resources (i.e., highly referred web sites). If the most useful resources for a user are not well-known, their ranking will be low; therefore it becomes time-consuming or unlikely to locate the most useful resources in the returned search results. That is, a user looking for specific information needs to mine and refine search results by spending considerable amount of time. Unfortunately, after refining search results, the refined results cannot be reused by other users looking for the same or similar information, because current search engines do not effectively enable users with similar interests to share their findings.

In this paper, we propose an agent-based approach for distributed semantic search. Our approach does not depend on the semantically annotated Web resources; instead
it depends on reusing the search results refined by users. In our approach, each user is represented by an agent, which has a local ontology to maintain useful search concepts. A search concept corresponds to a semantically described search interest. For the representation of search concepts, we use OWL-DL, i.e., Description Logic (DL) subset of Web Ontology Language (OWL). In this setting, searching the Web for a specific interest is formulated as finding instances of the related search concept. These instances may be any thing (e.g., web pages, videos, files and so on) that can be referred uniquely using a URL. When a user needs to search the Web for specific information, he semantically describes his search interest to his agent. If the agent does not know a search concept corresponding to this search interest, it communicates with other agents with similar interests to learn the concept or cooperatively create it. Once the agent has the search concept in its local ontology, it can retrieve the instances of this concept by querying its local ontology or the other agents. The queried agents are chosen among the ones that are believed to have similar search interests. After being queried, these agents find the most related search concepts in their local ontologies and send the known instances of these search concepts to the querying agent. If the querying agent does not get enough number of results from other agents for the search concept, it queries its user’s favorite search engines and lets the user select the most related resources. Then, these resources are recorded as instances of the search concept. Hence, the elicited results are later reused and shared with others that have similar search interests.

The rest of the paper is organized as follows. Section 2 and 3 explain how search concepts are described and compared. Section 4 proposes the way agents interact to evolve their ontologies, discover others and retrieve search results. Section 5 evaluates the proposed approach using simulations. Lastly, Section 6 discusses the proposed work with references to the literature and outlines directions for further research.

2 Describing Search Interests

In real life, people search the Web when they have a specific search interest. Even though two users have the same search interest, they may use different queries to describe it during their sessions with the search engines. Even a single user may need to use different queries to represent her specific search interest to a search engine. For example, she starts with some initial keywords, then refines or relaxes her queries depending on the returned results. She finds other keywords if she believes that the used keywords are not descriptive for the search engine. All of these queries can be thought as a projection of the same search interest in the pursuit of getting the most relevant search results from the search engine.

Search interests of users cannot be represented directly to the search engines during search sessions, because current search engines do not provide tools for users to explicitly and correctly represent their search interest. Hence, users try to come up with combinations of keywords to represent their search interests. Example 1 demonstrates how challenging it is to represent search interests using only keywords. Current Semantic Web technologies enable description of complex concepts using DL and makes reasoning on these concepts possible and feasible. Therefore, instead of using ambiguous keyword-based queries, we propose to use Semantic Web technologies to explicitly

1 http://www.w3.org/TR/owl2-overview
and clearly describe search interest of users. That is, each search interest is represented as a search concept using an OWL ontology and DL as shown in Example 2.

**Example 1** Assume that editor of a video website wants to add new videos to the website. He is specifically interested in “Comedy Videos that contain a muscular guy who is chased by a small dog”. In order to locate such videos, the editor needs to search the Web. Although his search interest is explicit, he cannot represent it clearly to the search engines. That is, he has to find the best combination of keywords that give the best results for his queries (e.g., small dog chase muscular man comedy). Consider another search interest that is about “Comedy videos that contain a muscular dog which is chased by a small man”. The keyword-based queries for these two search interests are expected to be identical or considerably overlapping. That is, keyword-based search engines may return the same videos for these distinct search interests.

**Example 2** Using the terms from an ontology, the search interest “Comedy Videos that contain a muscular guy who is chased by a small dog” is represented as Comedy $\sqcap$ Video $\sqcap$ $\exists$ contain.(Small $\sqcap$ Dog $\sqcap$ $\exists$ chase.(Muscular $\sqcap$ Man)). On the other hand, the search interest “Comedy videos that contain a muscular dog which is chased by a small man” is represented as Comedy $\sqcap$ Video $\sqcap$ $\exists$ contain.(Small $\sqcap$ Man $\sqcap$ $\exists$ chase.(Muscular $\sqcap$ Dog)).

In this paper, we envision a multi-agent system that consists of agents representing human users. Each agent has access to a common meta-ontology that contains primitive concepts and properties (e.g., concepts like Man, Dog, Video and so on). This ontology is static and does not contain any search concept. It constitutes grounding for describing search concepts and sharing this description between the agents. In addition to the common ontology, each agent has a local ontology, which contains the search concepts known by the agent. Each search concept is described using only the concepts and the relations from the common ontology. When an agent makes descriptions of its search concepts available to others, other agents can interpret and reason about these search concepts easily using a DL reasoner, such as Pellet [9].

Each agent has a unique identifier such as a URI and a unique namespace. For example, the agent of John Doe has a unique identifier http://agent.johndoe and its namespace contains every name starting with this URI. When the agent creates a new search concept such as ComicDogVideos, it gives a name to this search concept within its namespace such as http://agent.johndoe/ComicDogVideos. In this way, name conflicts between the search concepts that are created by different agents are prevented.

### 3 Similarity of Search Concepts

Each search concept is defined using the concepts from the common ontology. Hence, given a set of search concepts, an agent can reason about the relationships between these concepts using a DL reasoner. For example, Table 1 lists five different search concepts and their descriptions. Relationships between these concepts can be computed by an agent as follows. First, the agent creates an empty OWL ontology that imports the common ontology. Second, this ontology is populated with the search concepts in Table 1; each concept is created in the ontology using its DL description. Third, a reasoner is used to compute inferred taxonomy of the concepts within the ontology. The inferred taxonomy for the concepts in Table 1 is shown in Figure 1.
### Table 1. Some search concepts and their descriptions.

<table>
<thead>
<tr>
<th>Concept Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComedyVideos</td>
<td>Comedy ⊓ Video</td>
</tr>
<tr>
<td>GYMComedyVideos</td>
<td>Comedy ⊓ Video ⊓ location.GYM</td>
</tr>
<tr>
<td>ChasingComedyVideos</td>
<td>Comedy ⊓ Video ⊓ contain.(∃chase.⊤)</td>
</tr>
<tr>
<td>SmallDogChasingMuscularManVideos</td>
<td>Comedy ⊓ Video ⊓ contain.(Small ⊓ Dog ⊓ ∃ chase.(Muscular ⊓ Man))</td>
</tr>
<tr>
<td>SmallManChasingMuscularDogVideos</td>
<td>Comedy ⊓ Video ⊓ contain.(Small ⊓ Man ⊓ ∃ chase.(Muscular ⊓ Dog))</td>
</tr>
</tbody>
</table>

**Fig. 1.** Inferred taxonomy of search concepts in Table 1.

Taxonomy of search concepts provides valuable information such as subsumption relationships or similarity between the concepts. In a concept taxonomy, similarity between two concepts can be estimated by calculating the distance between these concepts. The length of the path between any two concepts indicates how similar these concepts are. There are various distance-based semantic similarity metrics [2, 8, 10], which can be used by the agents. In our experiments, we have used the similarity metric proposed by Wu and Palmer [10], because of its intuitiveness and simplicity. Accordingly, the agents compute similarity between \( c_1 \) and \( c_2 \) using Equation 1. Let \( c_0 \) be the most specific concept subsuming both \( c_1 \) and \( c_2 \). In the equation, \( N_1 \) is the length of the path between \( c_1 \) and \( c_0 \); \( N_2 \) is the length of the path between \( c_2 \) and \( c_0 \); lastly, \( N_0 \) is the length of the path between \( c_0 \) and the root of the taxonomy.

\[
sim(c_1, c_2) = \frac{2 \times N_0}{N_1 + N_2 + 2 \times N_0} \tag{1}
\]

An interesting property of distance-based similarity metrics is their instant response to the changes in the taxonomy. That is, while new concepts are added to or existing concepts are removed from a concept taxonomy, similarity between two specific concepts in the updated taxonomy may change immediately. This is simply because of the fact that, in the updated taxonomy, the distance between these two concepts may increase or decrease after the insertions and deletions as illustrated. This enables agents to automatically incorporate their changing conceptualization of the world into semantic similarity calculations.

### 4 Interactions of Agents

When a user has a search interest, he interacts with his agent to describe his search interest using an interface. Then, the agent converts the described search interest into a DL representation. Using this representation and a DL reasoner (e.g., Pellet [9]), the agent searches its local ontology for a semantically equivalent search concept. If an equivalent search concept is found, direct and inferred instances of the found concept
are retrieved from the ontology. Each retrieved instance corresponds to a URL of a Web resource. Retrieved instances are returned to the user after ranked according to some metrics. If an equivalent concept is not found or instances of the found search concept are not enough, the agent interacts with other agents to locate Web resources related to the search interest.

In order to get information related to its current search interest, the agent interacts with its neighbors (see Definition 1). To select its neighbors, each agent models other agents in the society by keeping track of their search interests from previous interactions. This is achieved by keeping track of the search concepts used by others during interactions. The annotation property usedBy is used to associate search concepts with the agents that use these search concepts during their interactions. For a new search interest, the agent can determine its neighbors as follows. In order to represent its search interest, the agent creates a new search concept in its local ontology. The new concept is placed into the concept hierarchy using a reasoner. Then, the agent finds the search concepts most similar to the new search concept. Similarity between search concepts is computed as explained in Section 3. Once the most similar search concepts are determined, their usedBy property is used to identify neighbors. Then, the agent interacts with those neighbors.

**Definition 1** Let an agent \( A \) be interested in a search concept \( S \). Neighbors of \( A \) with respect to \( S \) are defined as those agents that are also interested in \( S \) or a similar search concept.

### 4.1 Emergence of New Search Concepts

Search concepts are formal descriptions of search interests. If we can consider the Web as a collection of ad-hoc resources, each search concept defines a proper subset of the Web. If users having the same or similar search interests can locate one another, they can collaboratively search for the best resources for their interest or simply share and reuse URLs of the already discovered resources related to their search interests. This can only be achieved if users share their search concepts with one another.

An agent creates a new search concept when its user has a new search interest and none of the known search concepts in its local ontology can completely describe the interest. We may note that the knowledge of the agent about existing search concepts in the society is limited. This means that the new concept may either already exist in the society but the agent may not be aware of it or the concept may be totally new to the entire society. In order to differentiate between these two cases, the agent sends a *Concept Inquiry Message* to its neighbors to find out if the search concept is already known to its neighbors. This message contains the description of the desired search concept and the unique identifier of the agent. Upon receiving the concept inquiry message, a neighbor inspects the search concepts in its ontology to find a semantic match with the desired search concept (a semantically equivalent concept) using the DL reasoner and informs the requesting agent if there is a match or not. If a semantic match is found, then the neighbor sends the name of the matched concept in its ontology to the requesting agent. Therefore, the agent can add the desired search concept into its ontology with this name. In this way, the agent and its neighbor address this search concept with the same name in their ontologies.
If the agent receives different names from its neighbors for the same concept inquiry message, it notes that these names are synonyms, because they refer to the same concept. If none of the neighbors locates the desired search concept within their ontologies, the agent concludes that this search concept is not known by any of its neighbors. In this case, the agent places the concept into its local ontology with a unique name. We may note that concept names are created within the namespaces of the agents. That is, it is not possible for two agents to create two different search concepts and give the same name to them. Therefore, each concept name is unique and associated with only one search concept in the agent society. By giving unique names to the new search concepts, we remove the probability of name conflicts.

At the end of the procedure above, the agent adds the new search concept into its local ontology. During its cooperation with its neighbors, the agent gathers important information about the search concept. The agent shares the gathered information with its neighbors by sending a Concept Consolidation Message. This message contains the description of the search concept, and its name in the agent’s ontology. It also contains the identifiers of the neighbors who already know the search concept and the names of this search concept within their ontologies (referred as ”synonyms”). When a neighbor receives a consolidation message, it may add the described search concept into its ontology with the referred name if its ontology does not contain the search concept yet. Furthermore, the neighbor may store the referred synonyms to remember how the same search concept is addressed by others. Therefore, the agent and the neighbors can understand each other during their future communications regarding this search concept.

In the proposed approach, when an agent generates a new search concept to represent its new search interest, it teaches this search concept to its neighbors by sharing the description of the concept or the neighbors inform the agent about the search concept if the concept is already known by them. This leads to an interactive learning of new search concepts. Hence, mutually understood search concepts emerge as a result of agents’ social interactions.

4.2 Discovering Others

The approach proposed in this paper depends on the social interactions of an agent with other agents who have similar interests. When an agent has a new search interest, it communicates with others that have used a similar search concept in their interactions. For example, if an agent interested in comedy videos where a dog chases a man, it may communicate with the agents who are interested in comedy videos about chasing. These agents are determined using the usedBy annotation property attached to the known search concepts. In many cases, the agent may need to expand its knowledge about the society by discovering new agents with a specific search interest. In order to get the identifiers of the agents with a specific search interest, the agent generates a Peer Discovery Message. This message contains the identifier of the message originator, the name of the search concept that represents the search interest, the desired number of results that should be returned by the receiver and lastly a time-to-live (TTL) value to define how long the message should be forwarded. Then, using its local ontology, the agent chooses a subset of its neighbors to whom the message will be sent as explained in Section 4.1.
When another agent receives this message, it checks whether the search concept in the message is known or not. If this concept is not known, the receiver requests its description from the message originator. The receiving agent processes the message as follows. First, it computes the similarity of the search concept in the message to other search concepts in its ontology. Then, it sends the identifiers of the agents associated with the most similar concepts along with the concept names to the originator of the peer discovery message. The message originator updates its ontology using these entries. The receiver decides the number of entries to be sent using the number defined in the message. The receiver also updates its ontology by associating the name of the message originator with the related search concept using the `usedBy` property. As a result, the receiver remembers the search interest of the message originator and uses this information in the future. If TTL value of the message is greater than one, the receiver decrements the TTL value of the message and forwards the message to its neighbors that are most related to the search concept in the message.

Using these simple interactions, agents learn search interests of one another and update their ontologies. During the interactions of the agents, if unknown search concepts are encountered, the agent may request for the description of these concepts from the agents that have used these concepts in their interactions. Then, these concepts can be added to the ontology of the agent if they are valuable for the agent. Local ontology of the agent may grow rapidly as new concepts are added over time. As a result, ontological reasoning may become inefficient. To handle this, the agent may add learned search concepts into its ontology only if these search concepts are related to its own search interests. Similarly, the agent may remove rarely used search concepts from its ontology.

### Algorithm 1 Retrieving Search Results

1: **Input**: ConceptDescription \( D \), Ontology \( O \), Threshold \( \alpha \)  
2: **Output**: ResultSet \( R \)  
3: \( R = \{\emptyset\} \)  
4: Concept \( C = \) findEquivalentConcept\((D,O)\)  
5: if \( (C == null) \) then  
6: \( C = \) createNewSearchConcept\((D,O)\)  
7: end if  
8: \( R = \) getInstances\((C)\)  
9: if \( (|R| < \alpha) \) then  
10: queryNeighborsForInstances\((C)\)  
11: \( R = \) getInstances\((C)\)  
12: end if  
13: Rank\((R)\)

### 4.3 Retrieving Search Results

Using the approaches introduced in the previous sections, an agent can retrieve search results for a specific search interest. For this purpose, the agent follows the steps that are summarized in Algorithm 1. First, the agent queries its local ontology for a search concept corresponding to its current search interest (line 4). Note that the agent elicits the description of the search interest from the user (e.g., through an interactive user interface) and converts it into DL formalism using the common meta-ontology. Hence,
using a DL reasoner, it is straightforward to discover concepts semantically equivalent
to the description. After reasoning about the search concepts in its local ontology, if
the agent determines a search concept \( C \) semantically equivalent to the description of
the search interest (line 4), the agent uses the determined concept to address the search
interest. On the other hand, if none of the know search concepts corresponds to search
interest, a new search concept is created and added into the local ontology as explained
in Section 4.1 (lines 5-7). During the creation of the new concept, the agent does not
only learn related search concepts from its neighbors, but also discovers other peers that
have the same or similar search interests.

Based on its DL description, the place of the new concept \( C \) in the concept hierar-
chy is inferred by the DL reasoner [9]. Given the concept \( C \) from its local ontology, the
agent retrieves direct and indirect instances of \( C \) (e.g., instances of \( C \)'s sub-concepts)
from its local ontology using the reasoner and populates a set \( R \) with those instances
(line 8). If the agent currently does not know enough number of Web resources related to
the search interest, the size of the set \( R \) would be less than the desired number of search
results, which is defined by the user through the threshold \( \alpha \) (line 9). In this case, in
order to learn other instances of the search concept, the agent sends an \textit{Instance Re-
quest Message} (IRM) to its neighbors, which may be knowledgeable about the search
concept (line 10). An IRM contains the name of the search concept \( C \) as well as the
maximum number of instances that should be returned (\( n_{\text{max}} \)). If the concept name in
the IRM is not recognized by a neighbor, description of the concept is requested and
if desired, the concept is added to the neighbor’s local ontology with the related in-
formation. When a neighbor receives an IRM, it tries to locate the related search concept
\( C \) and \( C \)'s sub-concepts in its local ontology. Then, the neighbor rank the instances of
these concepts and sends the best \( n_{\text{max}} \) instances and their type to the sender. We may
note that all instances of \( C \)'s sub-concepts are also valid instances of \( C \). Therefore, even
though the neighbor does not have \( C \) in its local ontology, it may send the instances of
\( C \)'s sub-concepts that exist in its ontology. These sub-concepts are determined using the
description of \( C \) and ontological reasoning. In an IRM query, the number of instances
that should be returned by an agent is limited to prevent vulnerability of the system to
abuse and motivate correspondent agents to return the most genuine instances of the re-
lated search concept [5]. While it is not in the scope of this paper, the returned instances
can be used by an agent to measure the reliability of the correspondent agents. Different
approaches have been proposed in the literature to determine reliability of an agent
depending on the information it provides to others [7]; one of these approaches (e.g.,
the one proposed in [4]) can be utilized to compute the reliability of the corresponding
agents. The computed reliability of the agents can be used while selecting the neighbors
or ranking the search results before presenting to the user.

As a result of IRM query, the agent learns new instances of \( C \) and populates its
local ontology with these instances. Then, \( R \) is populated with the direct and inferred
instances of the search concept \( C \) (line 11). Lastly, the instances stored in \( R \) is ranked
before presenting them to the user (line 13). The ranking may depend on many factors;
some of them can be listed as follows.

- **Semantics:** Some instances in \( R \) may not be direct instances of \( C \). Let \( I \) be an
  instance of the concept \( C_i \), which is known to be a descendant of \( C \). Although \( I \) is
an inferred instance of $C$, its ranking depends on the semantic similarity between $C$ and $C_i$, i.e., it may increase parallel to the similarity.

- **Trust**: Let $I_X$ and $I_Y$ be instances of $C$ and learned from agent $X$ and agent $Y$, respectively. If the agent $X$ is more reliable than the agent $Y$, the ranking of $I_X$ should be higher than that of $I_Y$. Trust of one agent to another may depend on the quality of information previously provided [7].

- **Referrals**: Some search engines use referrals (in-links) to a Web resource to compute its rank (e.g., PageRank). A similar approach can be used to rank instances of $C$. Note that each instance $I$ of $C$ is a URL addressing a Web resource. Therefore, the number of referrals to $I$ from other Web resources (e.g., Web pages) can be used as a metric to calculate its ranking. In order to get these referrals, existing Web tools can be used. For example, Yahoo’s Site Explorer can be used to get referrals to a specific URL. The more referrals a Web resource has, the higher its ranking is.

Given an agent with a specific search interest, the proposed approach leverages the possibility that some users in the society may have already searched the Web for the same or similar search interests. Hence, if the agent can locate agents of these users, it can retrieve URLs of the most related Web resources from them. However, in many settings, the retrieved results from other agents may not be enough (e.g., during bootstrapping). If this is the case, the user may be prompted to search the Web using conventional tools such as search engines. Although enough number of the Web resources about the search interest is already known, the agent may also encourage its user to use those tools in order to enable better exploration of the Web. As the user discovers new Web resources for his search interest, the agent updates its local ontology with the discovered resources.

### 5 Evaluation

In order to evaluate our approach, we designed experiments\textsuperscript{2} as follows. First, we have created a meta-ontology\textsuperscript{3} for the description of search interests as explained in Section 2. This ontology introduces 65 concepts and 5 properties, as well as imports concepts and properties from the Food\textsuperscript{4} and Wine\textsuperscript{5} ontologies of W3C. This ontology can be further extended by importing other ontologies. Using the concepts from the meta-ontology, we created 600 different search interests. Then, we determined the relationships between these search interests using ontological reasoning and grouped the most semantically related search interests into roles. In this way, we have created 60 distinct roles; each role contain a number of search interests that are semantically related (i.e., subsumption and sibling relationships) as illustrated in Example 3.

**Example 3** If a role contains search interests related to Comedy $\sqcap$ Video, then the search interests referred by the concepts in Table 1 and Figure 1 are all in the scope of this role. That is, the users playing this role may search the Web for the instances of ComedyVideos, GYMComedyVideos, ChasingComedyVideos, SmallDogChasingMuscularManVideos, and so on.

\textsuperscript{2} The experiments are repeated 5 times and their mean is reported.

\textsuperscript{3} http://www.csd.abdn.ac.uk/~murat/websearch.owl

\textsuperscript{4} http://www.w3.org/TR/2003/CR-owl-guide-20030818/food

\textsuperscript{5} http://www.w3.org/TR/2003/CR-owl-guide-20030818/wine
We conducted experiments using 100 search agents that represent and interact with their users. Each agent is randomly given an initial acquaintance so that the underlying network of agents is connected. These agents use Pellet [9] for DL reasoning. Because the number of the agents is low, each agent is allowed to send peer discovery messages to at most three peers (e.g., neighbors) and TTL field of these messages is set to one; also the parameters $\alpha$ and $n_{max}$ are set to 10 and 5 respectively. As frequently done in multi-agent systems literature [11], we use synthetic users, which imitate human searching on the Web, to evaluate our approach. For a given search interest described in DL, a synthetic user can derive keywords from the description and form keyword-based queries. The user can send these queries to Google and decide which of the returned URLs are relevant to his search interest. Unlike a human user, a synthetic user cannot examine a Web resource (e.g., Web page, Video, and so on) to decide if its relevant to his interests. In order to imitate this, we created an oracle that knows which URLs are relevant to which search interests (see Definition 2). In order to decide on whether a URL relevant to his search interest or not, a synthetic user can query the oracle. On the average, only 10% of the URLs returned by Google are regarded relevant to a specific search interest by the oracle. In our study, a user browses only a limited number of links in the returned search results. This number is chosen randomly by the user between 10-100. The user stops examining the search results returned by Google if he finds five URLs relevant to his search interest.

**Definition 2** An oracle is a computational entity that decides if a specific Web resource is relevant to a specific search interest. The oracle is queried with $\langle D, url \rangle$, where $D$ is the DL description for the search interest and $url$ is the URL for the resource. The oracle has a look-up table containing entries of the form $\langle D_X, S_X \rangle$, where $D_X$ is the DL description of a search interest $X$ and $S_X$ is the set of URLs belonging to the resources relevant to $X$. When queried with $\langle D, url \rangle$, the oracle tries to find a table entry $\langle D_Y, S_Y \rangle$ such that $url \in S_Y$ and $D$ subsumes $D_Y$ ($D_Y \subseteq D$). If such an entry is found, the oracle returns $true$; otherwise it returns $false$.

For each experiment, we randomly select 10 roles and assign these roles to the users so that each user has one role. During an experiment, a user changes his role with a probability 0.05. Each experiment is divided into 10 discrete time window, each of which is called an epoch. At each epoch, a user tries to find URLs related to a randomly selected search interest from his current role. With a probability 0.25, the user searches the Web personally using Google; otherwise, the agent of the user takes over the searching task and try to find URLs using the proposed method. During this process, the agent determines a search concept that represents the search interest. If such a concept is not known by the agent, it is learned from others or created. The agent interacts with other agents to find the most relevant URLs using the proposed approach. The found URLs are declared as instances of the search concept representing the search interest. If the agent cannot determine enough number of URLs for the specified search interest, it prompts its user to search the Web for the related URLs. After determining the most related URLs, the user informs the agent and the agent adds these URLs as instances of the search concept. Therefore, other agents looking for URLs related to a similar search interest can use the found URLs, as proposed in the paper. Here, we evaluate our approach in four steps: emerging search concepts, messaging overhead, search results, and network topology.
Emerging Search Concepts

During the experiments, new search concepts have emerged to represent search interest of the users. Using these search concepts, agents reason about Web resources and interact with one another. Therefore, the system is populated with search concepts. Note that agents may have different search interests and neighborhood, so their local ontologies may differ significantly. Local ontology of an agent contains search concepts learned from others and the ones created directly by the agent. If a learned search concept can semantically represent its search interest, the agent does not create a new search concept, but it uses the existing one to refer the search interest. Figure 2 shows three curves: the total number of search concepts created in the environment, average number of search concepts in the ontology of an agent, and average number of search concepts in an agent’s ontology that were created by the agent itself. The first curve implies that at the end of 10 epochs, there are 160 search concepts created by the agents collaboratively. The second curve implies that, although there are 160 search concepts in the environment, an agent knows at most 20 concepts on the average. That is, the agent does not learn every search concept created in the environment, but the ones that are most related to its search interests. Interestingly, the third curve implies that at most 5 out of 20 search concepts are created by an agent itself, which means that most of the known search concepts are learned from others.

Later at this section, we will examine the emergent topology of the agent network and show that the interactions between agents lead to highly connected virtual communities (clusters), in which agents have similar search interests (i.e., play the same role). The resulting communities represent different search interest themes; that is, useful search concepts created in a community may be useless in another one. Therefore, agents tend to add new search concepts created in their community to their ontology, instead of learning the search concepts created in other communities. Interestingly, even though the ontologies of agents in the system are becoming highly different as new search concepts are created, ontologies of the agents in the same community are becoming more similar over time. It seems that agents in each virtual community create their own jargon to efficiently communicate regarding their common interests.

Messaging Overhead

Our approach is based on the interactions of agents through messaging. There are four

---

6 Note that, experiments are run for 10 epochs. In the figures, results for the first epoch is shown at epoch 1. Therefore, epoch 0 refers to time before the first epoch starts.
different message types in the system: peer discovery (PDM), instance request (IRM), concept inquiry (CIM), and concept consolidation (CCM) messages. Figure 3 shows how the total number of messages circulated in the system changes over time. At the beginning, the number of PDM messages increases because the agents try to discover others with similar search interests. However, after epoch 3, number of PDM messages start decreasing. The number of IRM messages increases over time, because agents find more neighbors to request URLs relevant to their search interests over time. Number of CIM and CCM messages decreases over time, because agents learn new search concepts from their neighbors over time, instead of creating them on their own. In overall, the total number of messages circulating in the system at any epoch is not significantly high with respect to the number of agents; moreover, it decreases over time. Hence, our approach does not flood the network exhaustively with messages.

**Search Results**

In the proposed approach, agent technology is combined with semantic Web technologies to share URLs classified by human users. For a user, there are two different ways of finding URLs relevant to his search interests. The first one is finding relevant URLs manually using search engines like Google. This process may be very exhaustive for the user in real life, because it requires the user to examine many URLs. The second one is learning the relevant URLs from the other agents in the community. Unfortunately, if an agent cannot learn enough number of relevant URLs from other agents, its user may search the Web manually and inform the agent about the found URLs. Figure 4 reports on the average how many of the relevant URLs learned from an agent’s user, or its peers. As shown, only a small amount of relevant URLs are learned by an agent directly from its user. This number is only 4.0 at the end of the first epoch, and rapidly decreases to 1.2 after a while. On the other hand, average number of relevant URLs learned from other agents increases from 0.5 to 21. This means that, agents successfully discover others with similar search interests over time and make use of the URLs previously refined by the users having same or similar search interests.

**Network Topology**

Agents start with a random connected network because of their initial acquaintances. After a while, they learn others with similar search interests and they communicate only with those. Hence, the overall network topology changes significantly. In this part of our evaluations, we examine how the network topology of agents changes over time as a result of their interactions. For this purpose, we have used the following metrics:

1. **Network diameter**: The longest shortest path across all pairs of nodes in network.
2. **Average distance between nodes**: Average length of shortest paths between all pairs of nodes in network.

3. **Average degree of nodes**: Average number of links that a node has in network.

4. **Average clustering coefficient**: This measure assesses the degree to which nodes tend to cluster together. It is computed by averaging the clustering coefficients of nodes in the network. The clustering coefficient of a node is the fraction of the node’s neighbors that are also neighbors of one another.

Figure 5 shows how the diameter and average distance between nodes in the network changes. In the beginning, diameter of the network is 15, but it decreases significantly and becomes 3.3 at the end of 10th epoch. Similarly, average distance between nodes is 6.5 in the beginning, but it decreases to 3.1 at the end of first epoch and eventually becomes 1.9 at the end of the experiments. Figure 6 shows how the average degree of nodes in the network changes over time. In the beginning agents know only a few of other agents (initial acquaintances); therefore, average degree of the nodes in the agent network is only 2. However, agents learn others with similar search interests over time and average degree of nodes in the underlying agent network increases up to 11.2. Lastly, we examine the clustering coefficient of the nodes in the underlying agent network. Figure 7 summarizes our findings. In the beginning, the underlying agent network does not have any clustering structure; so the average clustering coefficient is 0.0 at epoch 0. However, as the agents interact and discover others with similar search interests, interest-based clusters emerged over time. That is why the average cluster-
ing coefficient increases regularly and becomes 0.6 at the end of 10th epoch. We find that the network topology rapidly converges from a random network to a small-world network, with emerging clusters that match the user communities with shared interests.

6 Discussion
In this paper, we propose an agent-based approach for distributed semantic search on the Web. Our approach enables human users to describe their search interest to their agents. Our analyses of the proposed approach show that depending on the interests of their users, agents evolve their ontologies and create new search concepts. Based on these search concepts, the agents coordinate and compose virtual communities. As a result, not only agents with similar interests interact to find URLs relevant to their interests, but also shared vocabularies within virtual communities (i.e., jargons) are created cooperatively by agents to communicate properly.

Referral systems are proposed [11, 12] to cooperatively search information in social networks. A referral system is a multi-agent system whose member agents are capable of giving and following referrals. Each agent usually represents a user and they cooperate by giving and taking referrals to locate information relevant to the interests of their users. More specifically, when an agent needs specific information, it requests a set of its neighbors. The requested agents autonomously decide on providing directly the requested information, a set of referrals, or neither. As a result of their human-like interactions, agents compose dynamic social networks that lead to efficient and effective access to the desired information. However, unlike the proposed approach, current
referral systems do not allow agents to evolve their ontologies cooperatively and communicate using the shared concepts they have created.

If individual ontologies evolve on their own, agents have a major problem of ontology alignment. Aberer et. al. [1] propose an approach for the global semantic agreements. They assume that mappings between two different ontologies are already made by skilled human experts. These mappings are exchanged by the agents and global semantic agreements are reached using the properties of the exchanged mappings. On the other hand, in our approach, ontologies evolve cooperatively. As a result, not only useful search concepts emerge, but also local ontologies of the consumers having similar search interests become aligned over time. Moreover, our approach has a proactive nature; in our approach, an agent prevents future communication problems by informing its neighbors about the created concepts before using them.

In open environments like the Web, some of the agents may be malicious and distribute bogus information. In order to improve the robustness of the proposed approach, we plan to integrate techniques for agents to compute trustworthiness of their peers. Furthermore, some search concepts may be subjective (e.g., funny videos) and may be interpreted differently by different users (i.e., a video may be funny for a user, while it is not for another one). In these settings, trust information may also help agents to filter information from different-minded users [6].

References

Micro-agents on Android: Interfacing Agents with Mobile Applications

Christopher Frantz, Mariusz Nowostawski, Martin Purvis
Information Science Dept., University of Otago, New Zealand

Abstract. The comparatively recent move towards smartphones, and along with this new operating systems, such as Android, offers new potential to build mobile agent-based applications. Android gives applications access to a wide-ranging set of sensors and different communication channels – realizing the notion of nomadic computing – and has an inherently concurrent internal architecture based on loosely coupled components. This combination makes it particularly suitable for agent-based applications. Yet, it has several limitations: Android is not a multi-agent system on its own behalf and does not consistently employ loose coupling to give access to its capabilities.

In this work we address this situation by introducing our extension to the Android platform. We have ported our lightweight micro-agent framework to the Android platform and directly interfaced it with Android platform facilities. This offers mutual benefits: agent-based applications can access Android functionality in a loosely coupled and unified fashion, while allowing the developer to consistently think in an agent-oriented manner; Android can use the micro-agent platform as a lightweight middleware module to build distributed applications in a hybrid fashion.

We present our system architecture, called 'Micro-agents on Android' or 'MOA', and describe an example application using this approach as well as a performance benchmark. We further outline potential application areas and contrast it to existing approaches to build multi-agent applications on Android.

Keywords: multi-agent systems, mobile applications, agent organisation, benchmark, micro-agents, android

1 Introduction

The consideration of mobile devices in multi-agent systems has often been limited to the provision of a downsized derivate of the full multi-agent system implementation (for Java implementations typically targeting Java 2 MicroEdition (J2ME)), resulting in limited performance and a reduced feature set. One example for this is JADE-LEAP [8].

The current transition from feature phones to the increasingly popular smartphones shows a significant change of the potential to use of agent-based applications on mobile devices. Smartphones come with a capability set which is foreign
to regular stationary systems, such as a wide-ranging sensor set (e.g. accelerometers, gyroscope, video-camera, GPS) and various communication channels (such as Internet, SMS, Bluetooth) which makes nomadic computing a realizable possibility. The 'smartness' of applications on those devices doesn’t typically derive from sophisticated intelligent features, but instead from a meaningful combination of those new capabilities in a both flexible and efficient manner. Consequently, many applications on those devices have a mash-up character (for example in the context of location-based services) with a stress on reusable application components.

A promising approach to facilitate the smooth composition of those application elements is provided by the increasingly deployed mobile application platform Android [3] which enforces a modeling paradigm of asynchronously communicating loosely coupled application components.

This approach shows some similarity to the principles of multi-agent systems. However, Android does not fully relieve application developers from low-level aspects such as interacting with actual sensors or communication handling, and demands for an explicit handling of threads to avoid applications with poor responsiveness or performance problems. Thus apart from the wide-ranging functionality and increased computing power available, smartphones still demand careful software engineering and cannot afford straightforward translation of heavyweight agent concepts directly to them – even to those running Android.

We think that the lightweight efficiently communicating notion of micro-agents is a useful approach to provide a symbiotic advantage for either technology: they allow the efficient implementation of agent-based applications on Android-based mobile devices by

– providing an organisational model to structure application functionality,
– transparently interfacing with Android application components,
– offering better performance than Android’s built-in inter-component communication mechanism (which is to be shown in this work), and
– serving as a light-weight middleware towards Android application components to facilitate distributed applications.

To show this potential we first introduce Android’s concept of application components and the interaction mechanisms. Then we introduce our micro-agent architecture and show its potential to interoperate with Android. Later, we describe the architecture of ‘Micro-agents on Android’ (MOA). We describe an actual application based on MOA to demonstrate its use for application development. We point out potential application areas of MOA based on its flexibility and interfacing qualities. Finally we relate it to existing approaches to use agents on Android.

1 Android has increased its market share (for smartphone operating systems) to about 33 percent in Q4 2010 – from about 8.7 percent in Q4 2009 [5].
2 Android and Micro-agents

2.1 Android architecture and developmental principles

Android [3] is a Linux-based software stack and application execution environment for use on mobile devices and comes with a comprehensive set of libraries for aspects such as security and GUI development. Applications themselves, including the built-in ones such as the phone application, are composed of a dynamically linked combination of application components. Android defines four basic types of application components, namely: activities, services, broadcast receivers and content providers [1]. Activities run in the foreground, are rather short-running, present a user interface, and can directly interact with the user. Multiple activities can be composed to create more comprehensive applications (e.g. wizards). Services complement activities, as they run in the background and are relatively long-running. Broadcast receivers are started and run upon announced broadcasts (e.g. indicating system start or received SMS). Broadcast receivers can then start activities or services and are destroyed immediately after execution. Content providers maintain storage for specific data sets (e.g. the phone contacts) and allow access by other components.

Activation and communication between those components is done asynchronously via messages, called intents. Android interprets intents as an abstract request specification. As intents themselves represent a generic, dynamically typed data structure, they can hold arbitrary application-defined content and allow late runtime binding. This mechanism ensures loose coupling of various application components.

Intents can be sent either in an explicit manner, using the target component’s class name, or in a more elaborate implicit manner. Implicitly resolved intents can contain either of the following elements which allows matching them against applications registered with the Android instance. Those include Actions, which the target component needs to perform (such as calling (CALL)); Data, which are uniform resource identifiers (such as tel://7843982); and finally Categories, which indicate an alternative for target component resolution and describe characteristics of the target application (e.g. BROWSABLE indicates that the target activity can be invoked by a browser). Along with this, intents can encapsulate arbitrary data (so-called extras) passed as key/value-pairs.

Applications need to register intent filters in order to be resolvable by the intent resolution mechanism. The resolution mechanism is based – as introduced with intents before – on actions, categories and/or data types which can be combined to describe an application’s purpose.

Comprehensive information on Android’s architecture and details on the applications components can be found under [3].

---

2 The only exceptions are content providers. Intents are not used for the activation of content providers; in this case a content resolver is used.
2.2 The micro-agent concept and its implementation

Micro-agents are goal-directed, autonomously acting entities without a particular prescribed internal architecture. Still we expect the architecture to support the notion of hierarchical agent levels of abstraction—i.e., micro-agents may contain within their internal architecture other, more elementary micro-agents—and we expect interactions to be based on efficient asynchronous and/or synchronous message processing. Although the mentioned aspects find, in stronger or weaker sense, consideration in conventional agent systems, one key objective of micro-agents is to allow a consistent ‘modeling in agents’, even when ‘drilling down’ to the lowest level of implementation (e.g., primitive micro-agents wrapping external resources).

We see the multi-level modeling with the combined use of micro-agents and eventual heavier notions as the computationally rational approach to satisfy the key characteristics of Agent-Oriented Software Engineering (AOSE) [7], namely

- **Decomposition** of functionality down to an appropriately fine level of granularity,
- **Abstraction** by selective hiding lower levels of the agent organisation, and
- **Organisation** which consistently describes the overall structure.

Inasmuch resource constraints (such as memory and battery capacity) are of particular concern in the context of mobile computing (and continue to be a concern with the more powerful smartphones) we see micro-agents as a low-threshold and easy-entry approach to allow the contemporary use of agent-based technology on mobile devices.

Although micro-agents do not commit to a particular internal architecture type, they are goal-directed, engage in multiple conversations, are computationally cheap and put strong focus on efficient execution and interaction so as not to harm overall system performance; with micro-agents the choice to instantiate yet another agent should have limited impact on system resources but be a matter of modeling pragmatism. Functionality is composed of and embodied by a larger number of functionally small entities and different levels of granularity. Because the efficiency of communication is paramount, the architecture affords two levels of communication: synchronous communication via direct method calls bound at runtime and asynchronous message passing.

In order to clarify the organisational aspects of the micro-agent metamodel (shown in Figure 1), it is discussed before giving a brief description of the overall platform architecture of our platform which we call $\mu^2$. The micro-agent

![Fig. 1. Core relationships in $\mu^2$](image-url)
model is based on the KEA model [9] and identifies (micro-)agent and role as first-order entities with various specializations. Micro-agents play an arbitrary number of roles. Roles, however come in three first-level specialisations, namely Group Leader roles, Passive Roles and Social Roles. Passive roles allow the most simplistic agent implementations only providing synchronous inter-agent communication. Their execution is extremely efficient, and programmers can use these to implement low-level functionality, in places where more coarse-grained agent implementations would fall back to embedded object-oriented or structured programming approaches. Social roles, in contrast, communicate via asynchronous messages and allow long-running concurrent conversations. Those are the main concern of this paper. Group leader roles are specializations in connection with the organisation of micro-agents. By default, each agent is associated with at least one group, its primary group. The group leader role allows any micro-agent to create a group itself and manage so-called sub-agents. The only exception is the predefined SystemOwner agent, which is the only agent to be its owning group’s owner – a recursion termination condition for the emerging agent hierarchy. In consequence, a consistent hierarchical organisation of arbitrary depth can be modelled using agent-oriented abstractions, not only allowing the decomposition into agents but also the definition of abstraction layers (by hiding sub-agents beyond a given level). The modeling of functionality can then be handled to any degree of granularity.

MessageFilters which are a specialization of social roles, and are a helper construct to support the organisational modeling and functional decomposition into sub-agents by means of message-based delegation.Incoming messages on the super-agent are dispatched to registered message filters (which are sub-agents playing the Message filter role). Message filters then match messages against individual patterns and eventually process those.

However, message filters are just one mechanism in terms of which to handle the functional decomposition in a semi-automated manner; the application developer is free to model the decomposition by other means (e.g. explicit definition of agent/sub-agent relationships and in-code handling of functionality delegation). For all cases, though, agents will at least be sub-agents of the SystemOwner agent to allow consistent platform management.

Role implementations themselves register applicable intents which allow roles to be discovered for a dynamic binding of requests. In \( \mu^2 \) intents resemble the notion of intentions and include the information necessary for fulfillment. Intents in \( \mu^2 \) have a similar function as Android’s intents, but instead of a fixed method set (as with Android), intent type implementations are entirely left to the application developer. As a consequence, \( \mu^2 \) intents can have arbitrary structure (potentially including both properties and operations), which is not problematic, since only the requester and the executing agent need to know the semantics of the intent internals.

---

3 Patterns can be of a simple kind and merely test for particular message fields, or they can be more complex by taking the individual agent state into account.
Inasmuch as any role can register applicable intents (which its implementation needs to be able to handle) and also request the execution of intents, agent functionality can be composed across the entire agent organisation. Any execution request (sent via \texttt{send(intent)}) will receive a response either by the fulfilling agent or the platform agent in case of failed automated lookup of a potential target agent.

The event mechanism in $\mu^2$ follows the Publish-Subscribe pattern and is similar to the intent approach, but it requires the definition of the source of an event apart from the otherwise free implementation by the application developer. Once the event is raised all, agents (respectively their roles) subscribed to the event are notified. Intents, in contrast, are only delivered to one agent which is capable of fulfilling the intent (as determined by the platform).

The event subscription mechanism equally serves as a hook to react to system events (such as newly created agents, connecting platforms etc.). Both the intent-based dynamic binding and the raising of events works fully distributed across connected platform instances.

### 2.3 Comparison of $\mu^2$ and Android

The intent concepts highlighted in the previous sections differ in Android and $\mu^2$. While intents themselves are message containers in Android, $\mu^2$ introduces a separate message container (the MicroMessage) which encapsulates intents. Micro-agents do not necessarily rely on intents (if not using dynamic binding), and they can send any payload to an arbitrary agent (e.g. by addressing messages via agent name). Moreover, unlike Android’s intents MicroMessages allow for the specification of a sender.

One of the key facilities for a MAS is the ability to dynamically bind communication destinations. Android’s approach uses implicit intents which allow the lookup of registered \textit{intent filters} in order to invoke an application component. $\mu^2$, in contrast, looks up micro-agents playing roles which have registered their \textit{applicable intents}.

Overall, Micro-agents themselves can be seen as an equivalent to Android’s \textit{services}. Both have a lifecycle management and persist for longer periods of time (unlike the rather short-running \textit{activities} in Android). However, Android’s services do not show autonomous capabilities and are externally activated to provide their service. Additionally they do not engage in actual conversations. A key difference from the application modeling perspective is Android’s lack of an organisational model. Apart from the different application components Android does not provide modeling means for a structured application organisation.

\textit{Activities} in Android are similar to \textit{agent operations} which are not explicitly modelled in $\mu^2$ but are the result of interactions or are initiated to perform a specific task instance. In Android, activities have similar functionality but additionally provide a user interface.

The final concept to discuss is the concept of \textit{events}. Android reacts to external events via \textit{broadcast receivers} whose functionality can be implemented by the developer (e.g. sending intents to other activities). $\mu^2$ provides an explicit
subscription mechanism. Roles can subscribe to specific events and will receive an event object once raised. Both broadcast receivers and events can provide payload along with the notification.

Table 1 provides an overview on the similarities discussed to this point.

<table>
<thead>
<tr>
<th>Component</th>
<th>Android</th>
<th>(\mu^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message structure/container</td>
<td>Intent</td>
<td>Intent encapsulated in MicroMessage</td>
</tr>
<tr>
<td>Dynamic binding mechanism</td>
<td>Intent filter</td>
<td>Applicable intent</td>
</tr>
<tr>
<td>Persistent active entity</td>
<td>Service</td>
<td>Agent</td>
</tr>
<tr>
<td>Operation</td>
<td>Activity</td>
<td>Agent operation</td>
</tr>
<tr>
<td>Events</td>
<td>Broadcast receiver</td>
<td>Event subscription</td>
</tr>
</tbody>
</table>

Table 1. Comparison of components in Android and \(\mu^2\)

3 Micro-agents on Android

3.1 Design

The notion of loose binding both between micro-agents and Android’s application components motivates an integrated approach for mutual benefit: micro-agents enable an effective organisational modeling of agent-based applications on Android with unrestricted access to the device capabilities (sending SMS/MMS, retrieving GPS coordinates, accessing the phone’s address book). In addition to this micro-agents can potentially directly and spontaneously interact with existing applications or the user (e.g. to pick an address book entry) by raising according intents in the Android subsystem – supporting the principle and benefits of open systems.

Legacy Android applications, in return, can access micro-agent capabilities, respectively delegate functionality to micro-agents, or use the entire micro-agent framework as a middleware for distributed applications spanning across mobile devices as well as stationary devices running the desktop version of \(\mu^2\).

In order to realize this potential, \(\mu^2\) has been ported to Android as a first step. As a second step, the infrastructure of Android and \(\mu^2\) have been mapped against each other, constituting MOA. Its full schema is visualized in Figure 2 and explained in the following.

The interaction between Android and the micro-agents is mediated via a mutually linked micro-agent/service entity. Each of those two linked components (the AndroidInterfaceAgent and the MicroAgentInterfaceService in Figure 2) represents the interface to the according technological counterpart, i.e. for Android applications MOA appears as ‘yet another service’ whose lifecycle can be controlled from Android; micro-agents perceive Android as ‘yet another agent’ providing services (indicated by registered applicable intents) to other agents. The purpose of this combined agent/service is the dynamic conversion of requests/events raised from either side. Detailed differences handled by the conversion mechanisms are:
The respective intent class structures of µ2 and Android differ considerably.

Android’s intent invocation mechanisms require a specification of the application component type to be invoked (i.e. Activity (via `startActivity()`) or Service (via `startService()`)). In µ2 this is not of concern, as addressed entities are always agents.

Android’s intents (hereafter, for clarity, called `AndroidIntents`) do not maintain an explicit sender reference but invoke callbacks upon processing in the calling application component.

To match its more limited dynamic binding capabilities, an intent specialization rebuilding the `AndroidIntent` class method signature is provided with the micro-agent framework, which, in addition, allows the specification of the application component type. This particular intent type (AndroidExecutionIntent in Figure 2) is registered as an applicable intent with the interfacing micro-agent. Thus any intent to be raised in Android will consistently be resolved to this micro-agent. The Android service tied to this micro-agent additionally adds a custom sender field to any `AndroidIntent` and uses a dedicated activity (see IntentExecuterActivity in Figure 2) to raise all intents in Android. By implementing this Android activity’s `onActivityResult()` method, this IntentExecuterActivity it will be notified upon the processing of an intent. Maintaining the reference of the sender agent then allows eventual responses from an invoked application component to be forwarded to the original sender micro-agent (e.g. response to request for user input), thus overcoming the absence of a sender specification in Android. This allows micro-agents to formulate Android intents and thus directly interact with any other installed Android applications.

Since not all Android functionality – in particular hardware capabilities – can be addressed in an intent-based fashion, additional Utility components (which are either activities or services) are provided with MOA and are called from the interfacing service. Capabilities necessitating encapsulation in utility components

---

4 Figure 2 shows the SmsUtility and LocationUtility activity/service as examples.
are the different 'managers' which control Android capabilities (such as Location-
Manager to provide location information or SMSSManager which allows the sending
of SMS messages). Those utility components are complemented with Capability mapper micro-agents (e.g. MessagingCapabilityMapper, LocationCapabilityMapper) which register according applicable intents (e.g. SendSmsIntent, LocationRetrievalIntent, LocationSubscriptionIntent) and ensure the proper conversion to AndroidIntents as well as the interaction with the according utility component. Those capability mappers are modelled as sub-agents of the interfacing micro-agent, thereby exploiting the modeling capabilities provided by the micro-agent concept. Their registered applicable intents can be relatively simple, thus breaking down the Android capability into numerous specific intents or combining those into more coarse-grained intents.

From the Android side the InterfaceService can either be directly addressed via explicit intents (using the class name) or implicit intents which are resolved via according intent filters (represented as IntentFilter in Figure 2).

Another noteworthy aspect – and of particular of relevance when considering the context of nomadic computing – is the handling of system and/or application events. In order to receive events originated from the mobile device, such as receiving an SMS, the InterfaceService registers broadcast receivers (see BroadcastReceiver in Figure 2) for arbitrary events. The service itself forwards the message (via its micro-agent counterpart) to the corresponding capability mapper which raises an event in the micro-agent framework. Any subscribing agent will then receive the message (e.g. SMS message) via the event subscription mechanism of the micro-agent framework. In this way micro-agents can both address and receive intents/events from Android.

Expanding further on the potential symbiosis of MOA and Android, it is worth noting that Android applications can use MOA as a lightweight middleware for distributed Android applications, since MOA will dynamically dispatch application-dependent intents using its addressing mechanisms by sending an intent to the service and interpreting it as a distributed micro-agent event which ensures the dispatch to subscribing agents on any connected platform instance running on a mobile device or a desktop machine. This offers a significant infrastructural extension to Android, whose serialization per se is not fully compatible with the one of desktop JVM instances.

MOA’s dynamic binding approach in fact allows one to overcome further technological limitations. The current limited support for Just-in-time compilation on Android yet prevents the use of the programming language Clojure [6], a JVM-based LISP dialect which can be used for agent implementations in the desktop version of \( \mu^2 \). Using the dynamic binding mechanism (e.g. a ClojureExecutionIntent), Clojure code (e.g. received via SMS or entered by the user) can be automatically delegated to a potentially connected desktop platform version of \( \mu^2 \) and results returned upon execution. This enables the virtual execution of Clojure code on Android. Additional to the functional benefits realized by the interaction-centric dynamic linking associated with this approach is the nearly complete platform-independence of micro-agent implementations; they run both
with the desktop as well as the Android version of µ². Portability limitations (e.g. when relying on Clojure functionality) can be overcome using the cross-platform dynamic linking mechanism.

### 3.2 Context-aware assistant application

To get a better impression on designing MOA-backed Android applications, we describe one example application implemented using this approach. In this scenario an application receives an incoming message and must decide how to change the phone’s preferences in a location-sensitive manner and react to incoming calls or messages depending on the sender’s importance. A schematic overview is provided in Figure 3 and discussed in the following.

**Fig. 3.** Schematic Overview on location-aware mobile application

The application consists of a legacy Android activity used to maintain coordinates for locations of concern, such as ‘working place’, ‘main street’, ‘home’ and associates phone profiles. Additionally, another activity (ContactAnnotator) annotates contacts saved in the phone with priorities indicating their relevance to the phone owner (e.g. high priority for family members or boss). The storage locations for both data sets remain in the Android realm but are maintained by so-called content providers. Upon the start of the InterfaceService, MOA is started along with application-related agent implementations. In order to maintain actual agent implementations together with the MOA application components, an AgentLoader instance is passed to the InterfaceService which specifies the initialization of the micro-agents along with the Android application. The InterfaceService subscribes to proximity alerts for the entered coordinates. Once reached an according Event is raised (EnteredProximityEvent),
upon which the micro-agent managing phone profiles (PhoneProfileManager) activates the according phone profile (e.g. disable phone sounds and activate vibration in workplace environment). This includes the handling of potentially overlapping locations, in which case the more restrictive phone setting is chosen. An event which could also activate a particular profile (as an alternative to GPS-based location determination) could be connecting platform associated with a particular environment (e.g. workplace).

When receiving a phone call or text message, the InterfaceService is notified (Arc 1 in Figure 3), and the micro-agent receiving the incoming request decides, depending on the current profile (Interaction 3) and priority of the sender (Interaction 4/5a) how to react (e.g. if sounds are disabled in the workplace environment, call can be ignored, answered with automatic SMS promising a return call, or even override phone profile (for very important calls)).

In cases where the sender is not yet annotated (e.g. recently added), micro-agents can generate an Android intent to open an application-specific dialog (5b) in order to allow the annotation in real-time and handle the external request accordingly. This simple example shown here exemplifies the potential to delegate functionality to embedded micro-agents. Doing so provides structured agent-oriented modeling along with a flexible extensibility using elements such as synchronization of contact annotations with other connected phones, and consideration of calendar entries (on local phone and remote machines) to allow more precise responses where appropriate (“Am currently in a meeting with XY, ....”).

Key advantages of using micro-agents in conjunction with Android are:

- Consistent loose coupling – Micro-agents can address all Android capabilities in a unified loosely coupled manner; modelling of functionality is reduced to the mere composition of these intents. At no point do agent names need to (but can) be involved.
- Agent-oriented modeling – modeling applications using agent organisations with multiple levels of functionality granularity for maximum reuse. The flexible definition of intents and association with appropriate micro-agents allows mobile application developers to effectively specify their intent-based functionality repository while using the built-in organisation mechanisms to structure their applications.
- Distributed applications – MOA can be seamlessly distributed; developers do not need to deal explicitly with any network-related aspects.
- Performance – As briefly elaborated in the following subsection micro-agent interaction outperforms Android’s internal communication mechanism considerably, allowing effective decomposition into agent societies without performance loss.

An further consideration is the more extended realization of the open system principle, since agents can proactively interact with Android components which represent ’their’ environment.
3.3 Performance

To demonstrate the performance of Micro-agents on Android and their interaction mechanisms, we constructed a benchmark scenario for both AndroidIntents and micro-agent intents. The scenario is loosely based on the previously described context-aware application and was built as both a native Android application and as an application using the MOA approach. The internal functionality of agents and services is normalized to isolate the relevant comparative interaction performance. The scenario is shown in Figure 4 and described with references to the right section of the figure (Android scenario).

Fig. 4. Benchmark scenario for performance comparison

The \textit{BenchmarkService} (which is started via an Android activity) runs the given scenario for a specified number of rounds with the according benchmark variant (MOA or native Android).

The system simulates the arrival of a new SMS message by generating the initial intent which is dispatched to the ResponseManager (arc 1). The service identifies the SMS and requests the resolution of the received telephone number to a name (2) via the lookup in the phone’s contacts. Upon response (3) the SMS sender’s relevance is determined (arcs 4, 5). If the sender is of relevance (which is always the case in the benchmark), an SMS response is sent via the Responder (arcs 6, 7). The micro-agent version is functionally equivalent. It includes two additional messages as well as the conversion between Android intents and associated MicroMessages (with intent/event payload). As such, the interaction includes seven MicroMessages and two Android intents. In both cases the coupling of components is loose; intent filters (in the Android example) respectively applicable intents (with micro-agents) resolve target components/agents. Table 2 shows selected results of the benchmark.\footnote{The benchmark has been executed on an Intel Core2 Quad-Core CPU at 2.66 GHz and 3.25 GB RAM running on Windows XP Professional SP3 using the Android 2.1 Emulator. Each run has been undertaken 10 times and the results represent the average over the eight central results, ignoring highest and lowest values. In all the cases, we have allowed warm-up runs of 5 rounds before the actual timing.}

The results indicate a significant performance advantage of micro-agents over Android intents. Given those results, the use of micro-agents for interaction-
intensive applications is likely to be beneficial even if the multi-agent nature of
the application is not of primary concern for the application developer.

<table>
<thead>
<tr>
<th>Rounds</th>
<th>MOA (ms)</th>
<th>native Android (ms)</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>257</td>
<td>614</td>
<td>2.39</td>
</tr>
<tr>
<td>50</td>
<td>1834</td>
<td>4328</td>
<td>2.36</td>
</tr>
<tr>
<td>500</td>
<td>17757</td>
<td>42446</td>
<td>2.39</td>
</tr>
<tr>
<td>5000</td>
<td>156384</td>
<td>465606</td>
<td>2.98</td>
</tr>
</tbody>
</table>

Table 2. Selected Performance benchmark results per scenario rounds (with factors indicating relative performance of Android intents to micro-agents)

Android’s application components principle has potential support for low
d level system features (such as Inter-Process Communication (IPC) – which is not
used in this scenario). We consider the processing overhead associated with this
as one aspect introducing inefficiencies when executing intents. Conceptually,
micro-agents sit below the application component level, directly built from the
provided libraries, and thus do not share these performance penalties unless they
actively interact with Android application components.

3.4 Potential application areas

For a better impression of the potential of MOA, we outline areas where we deem
the presented approach promising, beyond the consideration of micro-agents as
lightweight middleware for distributed desktop/mobile applications.

Agent-based mobile applications
The use of MOA allows for the modelling and implementation of agent-based
applications which are compatible with desktop versions of the multi-agent plat-
form. This is of interest when considering the increasing computing power of mo-
bile devices. Furthermore, agent-based applications are not only restricted to run
on Android but to directly access device functionality and show the unique ca-
pability to natively interact with Android application components. Micro-agents
are particularly suitable for dealing more generically with heterogeneous application
environments and representing a low-threshold approach to the development
of smart mobile applications using agent-based principles without confining the
developer to a particular technology.

Intelligent agents
The original motivation for the micro-agent platform, i.e. to support the wider
software engineering advantages of agent-oriented decomposition, is equally avail-
able on mobile devices. The architecture-independent approach does not only
provide support for agent-based organisational decomposition and abstraction
levels, but also the ability to embed particular agent architectures (e.g. cognitive
architectures) which can then directly access phone capabilities (e.g. reading and
manipulation of contacts, calendar, SMS, etc), serving as a base for seamless inte-
gration of agent-based technology into mobile technology to enable intelligent
agent applications. Again, this can be realized on the devices or delegated to
more powerful desktop machines using MOA’s intent-based dynamic linking.

Robotics
Another area where the combined use of Android and agents is of practical con-
cern is the area of robotics. General-purpose robots with resource-limited computing capacity often only support a subset of necessary functionality (e.g. J2ME) and are relatively expensive. The use of MOA facilitates the embedding of the agent functionality in a standard Android device connected to the mechanical robot using wireless technologies (e.g. bluetooth). This "externalization" of the core agent capabilities could allow more economical use of robots and make upgrades a matter of replacing the phone. Additionally, MOA enables robots to communicate via SMS 'out-of-the-box' as a fallback mechanism if no other network connection is available (e.g. IP-based network) – or as general means of communication.

4 Related work

The use of Android in the context of agent-based applications is a very recent development. However, approaches to build agent-based systems in conjunction with Android have been undertaken by others which is briefly contrasted at this point.

Agüero et al. [4] present an approach to implement their Agent Platform Independent Model (APIM) in Android. The implementation is entirely based on the Android application components. Given that the elements of the APIM largely concentrate on agent internals, the modeling of agent organisations is not considered. Direct interaction with Android applications is not part of this concept.

JaCa-Android [10] is an implementation of the Agents and Artifacts model on Android. It identifies agents and artifacts as first-order entities to model agent-based systems. In order to implement agents, the approach embeds the Jason reasoning engine, thereby welding itself to a specific internal agent architecture. Relevant Android capabilities are encapsulated as artifacts which can be used by agents (e.g. GpsArtifact, SmsArtifact). This implementation provides an expressive means to handle Android capabilities using AgentSpeak, including the ability to act in remote workspaces. But agents in JaCa-Android cannot directly formulate Android-compatible intents to interoperate in a spontaneous manner.

A last notable approach to run agents on Android is the ported version of the multi-agent platform JADE [2]. It is conceptually weaker than any of the other approaches and represents the consequent move to provide a JADE version for Android. Its limitations notably include the requirement that a JADE main container provided by the full J2SE version of JADE (on a desktop system) is necessary for distributed operation. Apart from this only one agent can be run on a mobile device.

5 Conclusion

Android’s infrastructure shows characteristics which make it attractive for the development of multi-agent systems. However, its loose coupling does not extend to actual capabilities such as sensors. The integration of Android with micro-agents brings mutual advantages:
Loose coupling and Agent-based Modeling – Micro-agents can address Android capabilities and components in a consistently loosely coupled manner. Coarse-grained Android Services can be decomposed into micro-agents, which increases the reusability across different applications and remains computationally affordable as a consequence of the better interaction performance.

Distributed applications – Android can rely on MOA’s infrastructure to build distributed applications, across both mobile and desktop devices, allowing automated delegation of functionality.

Beyond this micro-agents further enable a hybrid approach by harmonizing with legacy application components; their unique ability to directly interact with other application components not only follows the spirit of open systems but also supports the development of mobile applications in a cross-paradigmatic manner.

High performance and distributedness make MOA a viable extension to the legacy mechanisms for general application development on Android, thus promoting agent-based concepts for the wider realm of software engineering. Plugging in specific agent architectures selectively will allow the integration of more intelligent features – beyond contemporary ‘smart’ applications.

References
Prognostic agent assistance for norm-compliant coalition planning

Jean Oh¹, Felipe Meneguzzi¹, Katia Sycara¹, and Timothy J. Norman²

¹ Robotics Institute, Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA 15213, USA
{jeanoh,meneguzz,katia}@cs.cmu.edu
² Dept. of Computing Science, University of Aberdeen
Aberdeen, UK
t.j.norman@abdn.ac.uk

Abstract. In this paper we describe a software assistant agent that can proactively assist human users situated in a time-constrained environment. We specifically aim at assisting the user’s normative reasoning—reasoning about prohibitions and obligations—so that the user can focus on her planning objectives. In order to provide proactive assistance, the agent must be able to 1) recognize the user’s planned activities, 2) reason about potential needs of assistance associated with those predicted activities, and 3) plan to provide appropriate assistance suitable for newly identified user needs. To address these specific requirements, we develop an agent architecture that integrates user intention recognition, normative reasoning over a user’s intention, and planning, execution and replanning for assistive actions. This paper presents the agent architecture and discusses practical applications of this approach.

1 Introduction

Human planners dealing with multiple objectives in a complex environment are subjected to a high level of cognitive workload, which can severely impair the quality of the plans created. For example, military planners during peacekeeping operations must plan to achieve their own unit’s objectives while following standing policies (or norms) that regulate how interaction and collaboration with Non-Governmental Organizations (NGOs) must take place. As the planners are cognitively overloaded with mission-specific objectives, such normative stipulations hinder their ability to plan to both accomplish goals and abide by the norms. We develop an assistant agent that takes a proactive stance in assisting cognitively overloaded human users by providing prognostic reasoning support. In this paper, we specifically aim to assist the user’s normative reasoning—reasoning about prohibitions and obligations.

In order to provide a user with a timely support, it is crucial that the agent recognizes the user’s needs in advance so that the agent can work in parallel with the user to ensure the assistance is ready by the time the user actually needs it. This desideratum imposes several technical challenges to: 1) recognize
the user’s planned activities, 2) reason about potential needs of assistance for those predicted activities to comply with norms as much as possible, and 3) plan to provide appropriate assistance suitable for newly identified user needs.

Our approach to tackle these challenges is realized in a proactive planning agent framework. As opposed to planning for a given goal state, the key challenge we address here is to identify a new set of goal states for the agent—i.e., a set of tasks for which the user will need assistance. After identifying new goals, the agent plans, executes, and replans a series of actions to accomplish them. Specifically, we employ a probabilistic plan recognition technique to predict a user’s plan for her future activities. The agent then evaluates the predicted user plan to detect any potential norm violations, generating a set of new goals (or tasks) for the agent to prevent those norm violations from happening. As the user’s environment changes the agent’s prediction is continuously updated, and thus agent’s plan to accomplish its goals must be frequently revised during execution. To enable a full cycle of autonomy, we present an agent architecture that seamlessly integrates techniques for plan recognition; normative reasoning over a user’s plan; and planning, execution and replanning for assistive actions.

The main contributions of this paper are the following. We present a principled agent architecture for prognostic reasoning assistance by integrating probabilistic plan recognition with reasoning about norm compliance. We develop the notion of prognostic norm reasoning to predict the user’s likely normative violations, allowing the agent to plan and take remedial actions before the violations actually occur. To the best of our knowledge, our approach is the first that manages norms in a proactive and autonomous manner. Our framework supports interleaved planning and execution for the assistant agent to adaptively revise its plans during execution, taking time constraints into consideration to ensure timely support to prevent violations. For a proof of concept experiment, our approach has been fully implemented in the context of a military peacekeeping scenario.

The rest of this paper is organized as follows. After reviewing related work in Section 2, we describe a high-level architecture of our agent system in Section 3. The three main components are described in detail in the following sections: Section 4 describes the agent’s plan recognition algorithm for predicting the user’s future plan; Section 5 describes how the agent evaluates the norm rules to maintain a normative state and to detect potential violations; and Section 6 presents how the agent plans and executes actions to accomplish identified goals. We present a fully implemented system in a peacekeeping problem domain, followed by other potential applications of this work in Section 7, and conclude the paper in Section 8.

2 Related work

This paper builds on previous work on assisting the user with information management [Oh et al., 2011] where the main discussion was on the algorithms for plan recognition and information management. In this paper, we specifically aim
at assisting the user with normative reasoning and autonomous planning and execution. Our paper is related to work in the literature on: 1) plan recognition, 2) assistant agents, 3) normative reasoning, and 4) norm monitoring.

2.1 Plan Recognition

Plan recognition refers to the task of identifying the user’s high-level goals (or intentions) by observing the user’s current activities. The majority of existing work in plan recognition relies on a plan library that represents a set of alternative ways to solve a domain-specific problem, and aims to find a plan in the library that best explains the observed behavior [Armentano and Amandi, 2007]. In order to avoid the cumbersome process of constructing elaborate plan libraries of all possible plan alternatives, recent work proposed the idea of formulating plan recognition as a planning problem using either classical planners [Ramírez and Geffner, 2009] or decision-theoretic planners [Baker et al., 2009]. The plan recognition algorithm used in our approach is based on a decision-theoretic planner, namely a Markov Decision Process (MDP) [Bellman, 1957].

2.2 Assistant Agents

An assistant agent is commonly modeled as a planning agent in literature. For example, Partially Observable Markov Decision Process (POMDP) models have been used to develop a hand-washing assistant for dementia patients [Boger et al., 2005] or a doorman assistant that helps a user navigate a maze by opening the doors [Fern et al., 2007]. These models are not suitable for a prognostic agent that must provide proactive assistance by predicting a user’s future actions in advance due to the following reason. In these approaches, the states of a POMDP are defined in terms of the variables describing both a user’s state and the agent’s state. Such models, however, do not include user actions (i.e., the actions defined in a POMDP are agent actions). Since the user actions are not represented in the models, it is not possible to predict future user actions. Moreover, as the number of states of a (PO)MDP grows exponentially in the number of state variables, these approaches suffer from the curse of dimensionality.

In contrast, we take a modularized approach. While an agent’s planning problem is defined using only those variables whose values the agent can directly modify, a plan recognition module keeps track of a user’s current and future activities to identify new tasks for the agent to prepare assistance. By separating the plan prediction from the agent’s action selection, our approach not only achieves an exponential reduction in the size of state space, but also enables the agent to simultaneously assist the user with multiple tasks because each new task is handled in an independent thread.

2.3 Normative Reasoning

In order to ensure that certain global properties of a society or organization are maintained, rules (or norms) that express permissions, prohibitions and
obligations have been developed [Jones, 1990]. Mathematical study of norms has been carried out in the context of deontic logic [von Wright, 1968], while computational treatment of these stipulations has been studied recently by the agents community as normative systems. These efforts led to the development of various formal models of norms [Vázquez-Salceda et al., 2005], as well as practical approaches to reasoning about norms within individual agents [Lopez y Lopez and Luck, 2003] and in a society [García-Camino et al., 2009]. The formalisms that allow modeling of norms for agent systems can also be used for the specification of the rules that humans must follow. Since this work is concerned with assisting a user to mitigate the cognitive load of planning under environmental norms, we leverage the formalisms to create an internal representation of the norms that the assistant must consider when providing assistance.

2.4 Norm Monitoring

In order for norms to be enforced in a norm-regulated system, various mechanisms were devised to monitor norm compliance within a system. The state of compliance of a set of norms within a system is known as the normative state [Farrell et al., 2005] and describes which agents are complying (or violating) which norms. Although various approaches to norm monitoring have been proposed [Farrell et al., 2005, Modgil et al., 2009, Hübner et al., 2010], they all rely on a deterministic logic view of the normative state. Without a probabilistic model of agent behavior, a norm monitoring mechanism can only assert whether a norm is definitely violated or not, lacking a gradual notion of how likely an agent is to violate a norm or when an agent is about to violate a norm. Thus, an assistant aiming to warn a user of potential violations can either constantly remind the user of all the norms in the system (which can potentially be violated), or inform the user after a violation has occurred that some remedial action should be taken. Whereas these approaches fail to address an important issue of prevention, our probabilistic normative reasoning approach allows the agent to detect probable norm violations in advance, preventing the user from violating norms.

3 Agent architecture

Figure 1 provides a high-level overview of our agent system for proactive yet unobtrusive assistance. The observer module monitors the user’s current activities in the environment to identify new observations that might indicate any changes in the user’s current and future plan. Given a new observation, the plan recognizer module uses a probabilistic algorithm to update its prediction for the user’s plans. From the predicted user plan, the norm reasoner module evaluates each plan step (actually the state resulting from these steps) to detect any potential norm violations. For each state in which norms are violated, the reasoner generates a new assistant task for the agent to carry out. The agent planner
module receives the new planning task that is to find a series of actions to prevent potential norm violations. When a prognostic plan is generated, the agent executes the plan until either the goal is reached or the goal becomes irrelevant to the user’s planning context. The effects of the successful plan execution can be presented to the user, e.g., notifying which actions the agent has taken to resolve a certain violation.

**Design assumptions:** A user’s planning state space is defined in terms of a set of variables and their domains of valid values, where a variable describes an environment and the progress status of certain activities. The variables are also used to specify regulating rules in the norm representation, e.g., a norm rule may define a relationship among a subset of variables. In general, such norm rules introduce additional variables to consider, adding extra dimensions into the reasoning process. As seen in a recent study [Sycara et al., 2010], when planning involves complex reasoning as in military environments, human users tend to lose track of policies (or norm rules), resulting in plans with significant norm violations. By developing an assistant agent that manages norm-related variables, our approach aims to relieve the user from having to deal with both task-specific variables and norm-related variables. We make a specific assumption that task-specific user variables and norm-specific agent variables are independent and thus changing an agent variable does not affect the values of user variables. For representation, let \((\text{user-variables}), (\text{agent-variables})\) denote a state composed of user variables and agent variables.

**Example scenario:** We use a simple example of peacekeeping scenario to illustrate the approach throughout the paper. We develop an assistant agent for a humanitarian NGO teamed with a military coalition partner. Consider a norm rule that an NGO must have an armed escort when operating in conflict areas. An escort can be arranged through a well-defined communication protocol, e.g., sending an escort request to and receiving a confirmation from a military party. Here, a state space can be defined in terms of two variables: area specifying the user’s geographic coordinates and escort indicating the status of an armed escort in each region. In our approach, a user can focus on reasoning about variable area only since the agent manages variable escort to assure that the user plan complies with norms. Note that variable escort is a simplified representation as
it is defined for each value of variable area, i.e., it is a function $\text{escort}(\text{area})$ to be precise.

In the following sections, technical details will be described for three main components: namely, plan recognizer, norm reasoner, and agent planner and executor. Due to space limitation, we omit the description for the observer and presenter modules that are responsible for interacting with the user.

### 4 Probabilistic plan recognition

Our plan recognition algorithm is leveraged from previous work [Oh et al., 2011]. We assume that a user’s planning problem is given as an MDP. Based on the assumption that a human user generally reasons about consequences and makes decisions to maximize her long-term rewards, we utilize an optimal stochastic policy of the MDP to predict a user’s future activities.

We compute an optimal stochastic policy as follows. Let $G$ denote a set of possible goal states. For each potential goal $g \in G$, we compute policy $\pi_g$ to achieve goal $g$ using a dynamic programming algorithm known as value iteration [Bellman, 1957]. Instead of a deterministic policy that specifies only the optimal action, we compute a stochastic policy such that probability $p(a|s,g)$ of taking action $a$ given state $s$ when pursuing goal $g$ is proportional to its long-term expected value $v(s,a,g)$ such that:

$$p(a|s,g) \propto \beta v(s,a,g),$$

where $\beta$ is a normalizing constant. The intuition for using a stochastic policy is to allow the agent to explore multiple likely plan paths in parallel, relaxing the assumption that a human user always acts to maximize her expected reward.

The plan recognition algorithm is a two-step process. In the first step, the algorithm estimates a probability distribution over a set of possible goals. For each observed state $s_t$ and action $a_t$ at time step $t$, the conditional probability $p(s_t,a_t|g)$ can be rewritten using Bayes’ rule as:

$$p(g|O_t) = \frac{p(s_1,a_1,...,s_t,a_t|g)p(g)}{\sum_{g' \in G} p(s_1,a_1,...,s_t,a_t|g')p(g')}.$$  

(1)
By applying the chain rule, we can write the conditional probability of observing the sequence of states and actions given a goal as:

\[ p(s_1, a_1, ..., s_t, a_t | g) = p(s_1 | g)p(a_1 | s_1, g)p(s_2 | s_1, a_1, g) \]
\[ \cdots p(s_t | s_{t-1}, a_{t-1}, ..., g) \].

We replace the probability \( p(a | s, g) \) with the user’s stochastic policy \( \pi_g(s, a) \) for selecting action \( a \) from state \( s \) given goal \( g \). By the MDP problem definition, the state transition probability is independent of the goals. Due to the Markov assumption, the state transition probability depends only on the current state, and the user’s action selection on the current state and the specific goal. By using these conditional independence relationships, we get:

\[ p(s_1, a_1, ..., s_t, a_t | g) = p(s_1)\pi_g(s_1, a_1)p(s_2 | s_1, a_1) \]
\[ \cdots p(s_t | s_{t-1}, a_{t-1}) \]. \hspace{1cm} (2)

By combining Equations 1 and 2, the conditional probability of a goal given a series of observations can be obtained.

In the second step, we sample likely user actions in the current state according to a stochastic policy of each goal weighted by the conditional probability from the previous step. Subsequently, the next states after taking each action are sampled using the MDP’s state transition function. From the sampled next states, user actions are recursively sampled, generating a tree of user actions known here as a plan-tree. The algorithm prunes the nodes with probabilities below some threshold. A node in a plan-tree can be represented in a tuple \( \langle t, s, l \rangle \) representing the depth of node (i.e., the number of time steps away from the current state), a predicted user state, and an estimated probability of the state visited by the user, respectively. Example 1 shows a segment of plan-tree indicating that the user is likely be in area 16 with probability \( .8 \) or in area 15 with probability \( .17 \) at time step \( t_1 \).

Example 1. \( \langle t_1, (\text{area} = 16), .8 \rangle, \langle t_1, (\text{area} = 15), .17 \rangle \)

5 Norm reasoner

In this section we specify the agent component responsible for evaluating predicted user plan to generate new goals for the agents. For this purpose, we utilize normative reasoning. Norms generally define constraints that should be followed by the members in a society at particular points in time in order for them to be compliant with societal regulations. We specify our norm representation format, followed by two algorithms for 1) predicting violations and 2) finding the nearest complying state towards which we can steer the user.

5.1 Norm representation

Inspired by the representation in [García-Camino et al., 2009], we define a norm in terms of its deontic modality, a formula specifying when the norm is relevant.
to a state (which we call the context condition), and a formula specifying the constraints imposed on an agent when the norm is relevant (which we call the normative condition). We restrict the deontic modalities to those of obligations (denoted $O$) and prohibitions (denoted $F$); and use these modalities to specify, respectively, whether the normative condition must be true or false in a relevant state. The conditions used in a norm are specified in terms of state variables and their relationships such as an equality constraint. Formally,

**Definition 1 (Norm).** A norm is a tuple $\langle \nu, \alpha, \mu \rangle$ where $\nu$ is the deontic modality, $\alpha$ is the context condition and $\mu$ is the normative condition.

**Example 2.** In the peacekeeping operations scenario, suppose that an intelligence message notifies that regions 3, 16 and 21 are risky areas. The norm, denoted by $\iota_{\text{escort}}$, that an NGO is obliged to have an armed escort can be expressed as:

$$\iota_{\text{escort}} = \langle O, \text{area} \in \{3, 16, 21\}, \text{escort} = \text{granted} \rangle.$$ 

**Definition 2 (Satisfiability).** A context condition $\alpha$ (alternatively a normative condition $\mu$) is satisfiable in state $s$ (so that $s \models \alpha$) if, for each variable $\varphi$ in state $s$, the current value assignment $v_{\varphi,s}$ of variable $\varphi$ in state $s$ is within the valid domain $D_{\varphi,\alpha}$ of variable $\varphi$ specified in condition $\alpha$, where the default domain is open if unspecified; such that $\forall \varphi \in s, v_{\varphi,s} \in D_{\varphi,\alpha}$.

### 5.2 Detecting violations

We say that a state is relevant to a norm if the norm’s context condition is satisfied in the state. When a state is relevant to a norm, a normative condition is evaluated to determine the state’s compliance, which depends on the deontic modality of the norm. Specifically, an obligation is violated if the normative condition $\mu$ is not supported by state $s$; i.e., $s \not\models \mu$. Conversely, as prohibitions specify properties that should not be realized, a prohibition is violated if the normative condition is supported by state $s$ such that $s \models \mu$. Formally,

**Definition 3 (Violating State).** Given state $s$ and norm $\iota = \langle \nu, \alpha, \mu \rangle$, a function determining the violation of norm $\iota$ in state $s$ is defined as:

$$\text{violating}(s, \iota) = \begin{cases} 
1 & \text{if } (s \models \alpha) \land (s \not\models \mu) \land (\nu = O) \\
1 & \text{if } (s \models \alpha) \land (s \models \mu) \land (\nu = F) \\
0 & \text{otherwise.}
\end{cases}$$

For instance, considering norm $\iota_{\text{escort}}$ in Example 2, given state $s = \{(\text{area} = 16), (\text{escort} = \text{init})\}$ the violation detection function $\text{violation}(s, \iota_{\text{escort}})$ would return 1, denoting that norm $\iota_{\text{escort}}$ is violated in state $s$.

Given a predicted user plan in a plan-tree, the norm reasoner traverses each node in the plan-tree and evaluates the associated user state for any norm violations. Recall from Section 4 that each node in a predicted plan-tree is associated with a user state and an estimated probability of the user visiting the node in the future. Using the estimated probability, the agent selects a set of high-risk norm violations to manage them proactively.
5.3 Finding the nearest compliant state

Our assistant agent aims at not only alerting the user of active violations but also proactively steering the user away from those violations that are likely to happen in the future. In order to accomplish this, for each state that violates a norm the agent needs to find a state that is compliant with all norms. That is, for each state \( s \) that violates any norm (i.e., \( \text{violating}(s, \cdot) = 1 \)), the agent is to find the nearest state \( g \) that satisfies all norms (i.e., \( \text{violating}(g, \ast) = 0 \)), where \( \cdot \) and \( \ast \) are regular expressions denoting any and all, respectively. Here, the distance between two states is measured by the number of variables whose values are different.

Since norm violations occur as the result of certain variables in the state space being in particular configurations, finding compliant states can be intuitively described as a search process for alternative value assignments for the variables in the normative condition such that norms are no longer violated, which is analogous to search in constraint satisfaction problems.

When a norm-violating state is detected, the norm reasoner searches the nearby state space by trying out different value assignment combinations for the agent-variables. For each altered state, the norm reasoner evaluates the state for norm compliance. The current algorithm is not exhaustive, and only continues the search until a certain number of compliant states, say \( m \), are found.

When compliant state \( g \) is found for violating state \( s \), state \( g \) becomes a new goal state for the agent, generating a planning problem for the agent such that the agent needs to find a series of actions to move from initial state \( s \) to goal state \( g \). The goals that fully comply with norms are assigned with compliance level 1.

When a search for compliant states fails, the agent must proactively decide on remedial actions aimed at either preventing the user from going to a violating state, or mitigating the effects of a violation. In the norm literature these are called contrary-to-duty obligations [Prakken and Sergot, 1996]. For instance, a contrary-to-duty obligation in the escort scenario can be defined such that if a user is about to enter a conflict area without an escort, the agent must alert the user of the escort requirement. For such partial compliance cases, we assign compliance level 2.

A planning problem can be expressed as a pair of an initial state \( s \) and a set of goal states \( g_i \) annotated with their compliance levels \( c_i \), such that \( \langle s, \{ (g_1, c_1), \ldots, (g_m, c_m) \} \rangle \).

Example 3 (Norm Reasoning). Given a predicted plan-tree in Example 1, if variable \( \text{escort} \) for area 16 has value \( \text{init} \) indicating an escort has not been arranged, the agent detects a norm violation and thus searches for a compliant state as follows. Let us define the domain of agent-variable \( \text{escort} \) to be: \{\( \text{init}, \text{requested}, \text{granted}, \text{denied}, \text{alerted} \)\}. By alternating values, we get the following two compliant states:

\[ \{(\text{granted}, 1), (\text{alerted}, 2)\}, \]

where state \( \text{granted} \) is fully compliant while state \( \text{alerted} \) is partially compliant from the agent’s perspective, as it complies with the contrary-to-duty obligation.
to warn the user. As a result, a newly generated planning problem is passed to the planner module as follows:

\[
\langle \text{init}, \{(\text{granted}, 1), (\text{alerted}, 2)\} \rangle
\]

6 Planner and executor

As opposed to precomputing a policy for every possible case, we propose a scalable model where the assistant agent dynamically plans and executes a series of actions as new problems arise. Note that the issues regarding adjustable autonomy are outside the scope of this paper. Instead, we use a cost-based autonomy model where the agent is allowed to execute those actions that do not incur any cost, but is required to get the user’s permission to execute costly (or critical) actions.

6.1 Planning

The agent has a set of executable actions. In the peacekeeping scenario, for instance, the agent has the following actions: \{\text{send-request, receive-reply, alert-user}\}. Given a planning problem—an initial and goal states—from the norm reasoner, the planner module is responsible for finding a series of actions to accomplish the specified goal. In Example 3, two goal (or absorbing) states have been assigned by the norm reasoner: an escort is granted or the user is alerted of the need for an escort. Thus, the agent needs to find a way to change the value of escort variable from \text{init} to either \text{granted} or \text{alerted}.

Since our representation of planning problems is generic, one may use classical planners in the implementation. Instead, we use an MDP to develop a planner in order to respect uncertainty involved in agent actions, e.g., sending a request may fail due to a communication network failure.

Recall that a predicted user plan from the plan recognizer imposes deadline constraints (specified as the depth of node) to the agent’s planning. Specifically, if the user is likely to commit a violation at a certain time step ahead, the agent must take actions to resolve the violation before the time step. In the planner, a deadline constraint is utilized to determine the horizon for an MDP plan solver, such that the agent planner needs to find an optimal policy given the time that the agent has until the predicted violation time.

In Example 3, when the violation is predicted far in advance, an optimal policy prescribes the agent to always request an escort from the other party, except if an escort request has been denied by the other party then the agent should alert the user of the denied request. Note that an optimal policy can change as time elapses, i.e., as the future horizon shortens, the expected values of states change. For instance, the user is better off by being warned when there is not enough time left for the agent to arrange an escort. We compare the number of sequential actions in a plan with the depth of node (or the goal’s deadline) to determine the plan’s feasibility.
The planning problem formulated by the reasoner may not always be solvable; that is, a compliant state can only be accomplished by modifying those variables that the agent does not have access to, or none of the agent’s actions has effects that results in the specified goal state. In this case, the agent notifies the user immediately so that the user can take appropriate actions on her own. Otherwise, the agent starts executing its actions according to the optimal policy until it reaches a goal state.

6.2 Execution

Execution of an agent action may change one or more variables. For each newly generated plan (or a policy) from the planner module, an executor is created as a new thread. An executor waits on a signal from the variable observer that monitors the changes in the environment variables to determine the agent’s current state. When a new state is observed the variable observer notifies the plan executor to wake up. The plan executor then selects an optimal action in the current state according to the policy and executes the action. After taking an action, the plan executor is resumed to wait on a new signal from the variable observer. If the observed state is an absorbing state, then the plan execution is terminated, otherwise an optimal action is executed from the new state.

In order to handle unexpected exceptions during execution time, an executable action has a timeout such that when the execution of an action reaches its timeout the plan is aborted. When a plan is aborted the specific goals of the plan have generally not been achieved. If the goals are still relevant to the user’s current plan (according to a newly predicted user plan), then the norm reasoner will generate them as new goals for the agent to accomplish.

The agent’s plan can be updated during execution as more recent assessment of rewards arrives from the norm reasoner, forcing the agent to replan. For instance, after the agent requested an escort from the other party, the other party may not reply immediately causing the agent to wait on the request. In the meantime, the user can proceed to make steps towards the unsafe region, imposing a tighter deadline constraint. When the new deadline constraint is propagated to the planner, an optimal policy is updated for the executor, triggering a new action, e.g., to alert the user of the potential violation (instead of trying to arrange an escort).

When alerted by the agent, the user may take certain actions to resolve the violation for herself, or alter her current plan to avoid the violation. In the worst case, the user may still proceed with the current plan and violate the norm. The agent is not penalized for such cases when we evaluate the agent’s performance.

7 Applications

Through this research, we aim to make not only scientific contributions but also practical impact on realistic applications. The autonomous assistant agent framework that has been presented can be applied to various problem domains. Here, we include some examples of potential applications.
7.1 Military escort planning in peacekeeping

As a proof of concept prototype, our approach has been implemented in the context of planning for peacekeeping operations, in a scenario inspired by the work in [Sycara et al., 2010], where two coalition partners (a humanitarian party Alpha and a military party Bravo) plan to operate in the same region according to each party’s individual objectives and a set of regulation rules.

Figure 2 shows a planning interface of a humanitarian party (Alpha). The figure is annotated with labels for illustration. At time step $T_1$, the agent identifies a norm violation at area 16 in the predicted user plan, for which the agent sends an escort request to Bravo. When the agent receives a reply from Bravo granting a permission the escort status is displayed in the interface. Similarly, the agent sends an escort request for area 21 for another norm violation, but Bravo does not respond. At time step $T_2$, an updated policy prescribes the agent to alert the user, and a warning is displayed in the interface.

We have used a simplified military escort planning scenario throughout this paper to illustrate our approach. In practice, the planning and scheduling of escort services in military peacekeeping operations involve complex norm reasoning due to diverse stakeholders. Through a series of collaborations with the military and various NGO groups, we have identified a significant amount of interest in developing software assistant for this problem domain, and we are currently working on scaling up the system to deal with more realistic settings.
7.2 Assistive living applications

It is important to note that the goal of this research is not to guide the user in finding optimal planning solutions, but instead, the agent aims to support the user’s plan by identifying and making amends for the plan’s weaknesses. As opposed to directing the user to make optimal decisions with respect to a certain objective (as in decision-support systems), we aim to design an agent that can maximize the support to help the user in making decisions based on her own criteria and judgement. From the user’s perspective, independent decision making is crucial in many problem domains such as assistive living technologies for the disabled and the elderly.

In this domain, the norm rules can be defined to specify a set of prohibitions for unsafe activities. When the agent predicts any potential dangers, the agent’s new goal becomes restoring a safe state. For instance, if the safe state can be accomplished by taking the agent’s available actions, e.g., moving certain objects on the floor, the agent can resolve the issue. When the agent cannot accomplish the goal using its own capabilities, the agent can instead alert the human assistant before an accident happens.

Similarly with the assistive living applications, our approach can also be applied to other care-giving applications.

8 Conclusion and Future Work

In this paper, we presented an assistant agent approach to provide prognostic reasoning support for cognitively overloaded human users. We designed the proactive agent architecture by seamlessly integrating several intelligent agent technologies: probabilistic plan recognition, prognostic normative reasoning, and planning and execution techniques. Our approach presents a generic assistant agent framework with which various applications can be built as discussed in Section 7. As a proof of concept application, we implemented a coalition planning assistant agent in a peacekeeping problem domain.

Our approach has several advantages over existing assistant agent approaches. When compared to other decision-theoretic models, our approach is significantly more scalable because of exponential state space reduction discussed in Section 2. As opposed to assistant agent models where an agent takes turns with the user, our agent has more flexibility in its decision making because the agent can execute multiple plans asynchronously. More importantly, our agent is proactive in that the agent plans ahead of time to satisfy the user’s forthcoming needs without a delay. Such proactive assistance is especially an important requirement in time-constrained real-life environments.

We made a specific assumption that agent variables are independent from user variables. We will investigate approaches to relax this assumption. Also, we will refine the algorithm for determining a plan’s feasibility in Section 6.1 by estimating expected time required for each action. Furthermore, we plan to extend our approach to work in a multi-user, multi-agent setting where resolving
a norm violation may involve multi-party negotiations. In addition, when there are more than one assistant agents, newly generated goals can be shared or traded among the agents. We will address these special issues raised in multi-agent settings in our future work.

Acknowledgments

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence and was accomplished under Agreement Number W911NF-09-2-0053. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References


