Seventh International Workshop on Programming Multi-Agent Systems

ProMAS'09

May 11-12, 2009
Budapest, Hungary

Preface

The ProMAS workshop series aims at promoting research on programming technologies and tools that can effectively contribute to the development and deployment of multi-agent systems. In particular, the workshop encourages the discussion and exchange of ideas concerning the techniques, concepts, requirements, and principles that are important for establishing multi-agent programming platforms that are useful in practice and have a theoretically sound basis. Topics addressed include but are not limited to the theory and applications of agent programming languages, the verification and analysis of agent systems, as well as the implementation of social structures in agent-based systems.

In its previous editions, since 2003 ProMAS constituted an invaluable occasion bringing together leading researchers from both academia and industry to discuss issues on the design of programming languages and tools for multi-agent systems. We are very pleased to be able to again present a range of high quality papers at ProMAS'09. Overall, in 2009 ProMAS received 34 submissions from which 9 were selected for long and 6 for short presentation. Due to the high number of quality submissions ProMAS could attract this year, it will be held, for the first time, as a 1.5 days workshop. Similar as in previous editions, the themes covered include a wide range of, theoretical, conceptual as well as technical aspects of multi-agent programming.

We hope that ProMAS'09 will continue the successful series of its previous editions and contribute to the vision of creating industrial strength programming languages and software tools for developing multi-agent systems.

May 2009
Lars Braubach, Jean-Pierre Briot, John Thangarajah
ProMAS.09 Workshop Organizers

Workshop Organization

ProMAS'09 is a workshop to be held on Monday 11th and Tuesday 12th May in Budapest, Hungary. The workshop is part of the AAMAS'09 Workshop Program.

Programme Chairs

Lars Braubach - University of Hamburg, Germany
Jean-Pierre Briot - LIP6, France
John Thangarajah - RMIT University, Australia

Steering Committee

Rafael Heitor Bordini - University of Durham, United Kingdom
Mehdi Dastani - Utrecht University, The Netherlands
Jurgen Dix - Clausthal University of Technology, Germany
Amal El Fallah Seghrouchni - University of Paris VI, France

Programme Committee

Matteo Baldoni - University of Torino, Italy
Guido Boella - University of Torino, Italy
Juan Botia Blaya - Universidad de Murcia, Spain
Keith Clark - Imperial College, United Kingdom
Rem Collier - University College Dublin, Ireland
Louise Dennis - University of Liverpool, United Kingdom
Ian Dickinson - HP Labs, Bristol, United Kingdom
Berndt Farwer - Durham University, United Kingdom
Michael Fisher - University of Liverpool, United Kingdom
Jorge Gómez-Sanz - Universidad Complutense de Madrid, Spain
Vladimir Gorodetsky - Russian Academy of Sciences, Russian Federation
Dominic Greenwood - Whitestein Technologies, Switzerland
James Harland - RMIT University, Australia
Koen Hindriks - Delft University of Technology, Netherlands
Benjamin Hirsch - TU-Berlin, Germany
Jomi Fred Hübner - ENS Mines Saint-Etienne, France
João Leite - Universidade Nova de Lisboa, Portugal
Viviana Mascardi - University of Genova, Italy
John-Jules Meyer - Utrecht University, Netherlands
David Morley - SRI International, United States
Jörg Müller - Clausthal University of Technology, Germany
Peter Novak - Clausthal University of Technology, Germany

Andrea Omicini - University of Bologna, Italy
Frederic Peschanski - LIP6 - UPMC Paris Universitas, France
Michele Piunti - ISTC - CNR and DEIS Università di Bologna, Italy
Agostino Poggi - University of Parma, Italy
Alexander Pokahr - University of Hamburg, Germany
Alessandro Ricci - DEIS, Università di Bologna, Italy
Ralph Rönnquist - Intendico Pty Ltd, Australia
Sebastian Sardina - RMIT University, Australia
Ichiro Satoh - National Institute of Informatics, Japan
Munindar P. Singh - NCSU, United States
Tran Cao Son - New Mexico State University, United States
Kostas Stathis - Royal Holloway, United Kingdom
Paolo Torroni - University of Bologna, Italy
Gerhard Weiss - SCCH GmbH, Austria
Wayne Wobcke - University of New South Wales, Australia
Neil Yorke-Smith - SRI International, United States
Yingqian Zhang Delft - University of Technology, Netherlands
Olivier Boissier - ENS Mines Saint-Etienne, France
Birna van Riemsdijk - Delft University of Technology, Netherlands
Leon van der Torre - University of Luxembourg, ILIAS, Luxembourg

Auxiliary Reviewers

Alferes, Jose
Bromuri, Stefano
Chopra, Amit
Gabaldon, Alfredo
Ghizzioli, Roberto
Hepple, Anthony
Kaiser, Silvan
Remondino, Marco
Torres, Viviane

Table of Contents

A MultiAgent System for Monitoring Boats in Marine Reserves 1
Giuliano Armano, Eloisa Vargiu

A Middleware for modeling Organizations and Roles in Jade 14
Matteo Baldoni, Guido Boella, Valerio Genovese, Roberto Grenna, Andrea Mugnaini, Leon van der Torre

Representing Long-Term and Interest BDI Goals 29
Lars Braubach, Alexander Pokahr

Debugging BDI-based Multi-Agent Programs 44
Mehdi Dastani, Jaap Brandsema, Amco Dubel, John-Jules Meyer

Space-Time Diagram Generation for Profiling Multi Agent Systems 59
Dinh Doan Van Bien, David Lillis, Rem Collier

An Open Architecture for Service-Oriented Virtual Organizations 74
A. Giret, V. JuliÃn, M. Rebollo, E. Argente, C. Carrascosa, V. Botti

Multi-agent systems: Modeling and Verification using Hybrid Automata . 89
Ammar Mohammed, Ulrich Furbach

Probabilistic Behavioural State Machines 103
Peter NovÃk

Golog speaks the BDI langauge 118
Sebastian Sardina, Yves Lesperance

Agent-Oriented Control in Real-Time Computer Games 133
Tristan Behrens

An Architecture for Multiagent Systems: An Approach Based on Commitments 148
Amit Chopra, Munindar Singh

A Computational Semantics for Communicating Rational Agents Based on Mental Models 163
Koen Hindriks, M. Birna van Riemsdijk

Introducing Relevance Awareness in BDI Agents 179
Emiliano Lorini, Michele Piunti

Modularity and compositionality in Jason 194
Neil Madden, Brian Logan

A Formal Model for Artifact-Based Environments in MAS Programming . 209
Alessandro Ricci, Mirko Viroli

Infrastructure for forensic analysis of multi-agent based simulations 228
Emilio Serrano, Juan Botia, Jose Manuel Cadenas

Author index 240

Keyword index 241

A MultiAgent System for Monitoring Boats in Marine Reserves

Giuliano Armano[1] and Eloisa Vargiu[1]

Dept. of Electrical and Electronic Engineering, University of Cagliari, Italy
{armano, vargiu}@diee.unica.it
http://iasc.diee.unica.it

Abstract. Setting up a marine reserve involves access monitoring, with the goal of avoiding intrusions of not authorized boats – also considering that typically marine reserves are located in not easily accessible areas. Nowadays, intrusion detection in marine reserves is carried out by using radar systems or suitable cameras activated by movement sensors. In this paper, we present a multiagent system aimed at monitoring boats in marine reserves. The corresponding scenario requires to discriminate between authorized and not authorized boats – the formers being equipped with GPS+GSM devices. Boats are tracked by a digital radar that detects their positions. The system has been used to monitor boats in a marine reserve located in the north Sardinia. Results show that adopting the proposed approach allows system administrator and staff operators to easily identify intrusions.

1 Introduction

Nowadays, the agent research community provides powerful theories, algorithms and techniques that may have a great potential in deploying various real applications. Several research and industrial experiences already put into evidence the advantages of using agents in manufacturing processes [15], web services and web-based computational markets [10], and distributed network management [6]. Moreover, further studies suggest to exploit agents and MASs as enabling technologies for a variety of novel scenarios, i.e., autonomic computing [14], grid computing [9], and semantic web [5].

According to [13], the main bottlenecks in fast and massive adoption of the agent-based solutions in real applications are: (i) limited awareness about the potentials of agent technology in industry; (ii) limited publicity of the successful industrial projects with agents; (iii) misunderstandings about the technology capabilities, over-expectations of the early industrial adopters and subsequent frustration; (iv) risk that comes with adoption of new technology that has not been already proven in large scale industrial applications, and (v) lack of design and development tools mature enough for industrial deployment.

In our opinion, multiagent technology may help in deploying real applications both from software engineering [17] and technological perspectives [4]. In this

paper we present a multiagent system aimed at monitoring boats in marine reserves. The system has been implemented by using X.MAS, a generic multiagent architecture devised to implement information retrieval and filtering applications [2].

The corresponding scenario requires to discriminate between authorized and not authorized boats – the formers being equipped with GPS+GSM devices. Boats are tracked by a digital radar that detects their positions. The system has been used to monitor boats in a marine reserve located in the north Sardinia. Results show that adopting the proposed approach allows system administrator and staff operators to easily identify intrusions.

The remainder of the paper is organized as follows: first, selected related work on agent-based systems for monitoring and surveillance are presented. Then, the underlying scenario is sketched. Subsequently, the adopted MAS solution is described focusing on the corresponding macro-architecture and on the implemented agent capabilities. The current deployed prototype is then presented. Conclusions and future research end the paper.

2 Related Work: Agent-based Systems for Monitoring and Surveillance

Application of agent methodologies in process monitoring and control is a relatively new approach, particularly suitable for distributed and dislocated systems. In particular, agent-based solutions have been proposed to monitor: (i) control systems [8], (ii) distant control experimentation systems [16], and forest fires [7].

In [8], a methodology for the design of agent-based production control systems, which can be successfully applied by an engineer with no prior experience in agent technology, has been proposed. Authors proposes a design method for identifying the agents of a production control system. The identification of agents allows the designer to move from pure domain concepts (such as production processes), to agent-oriented concepts (such as agents and decision responsibilities). In addition, the identification of agents provides the basis for all other subsequent design steps, such as interaction design or agent programming.

In [16], an approach to the realization of distant control experimentation system has been described, the main task being to realize the temperature and humidity monitoring and control experiment with the possibility of the video tele-presence (video monitoring) of the experimental area. The underlying idea is to integrate an existing laboratory greenhouse model and an existing video system. A multiagent system is then developed that involves five specific software agents. Agents are responsible of all operations: from user-system communication to telecontrol and video monitoring operations. As for the agent communication languages, KQML is used.

In [7], a TCP/IP-based system conceived of sensor networks, central server units for collecting, processing and storing all data, has been presented. Each sensor network has several monitoring units, each of them including: (i) pan, tilt, zoom-controlled video camera connected to the network-embedded video

web server, (ii) mini meteorological stations connected to network-embedded data web servers, and (iii) a wireless communication unit. Agents designed for the forest fire monitoring system follows these guidelines strictly, because the system is conceived as a modular system where each module is autonomous, aware of its environment and capable for active behavior if alarmed. Environment awareness is accomplished by connecting numerous meteorological sensors to a network-embedded microcontroller unit. Network-embedded microcontroller unit is responsible for collecting data from sensors, formatting and preprocessing it and giving it to the central server agent when asked for.

Agent-based solutions have also been proposed to develop video surveillance systems ([12], [1], [11]). Video surveillance is an active area of research. In this field, researchers have been concentrated on detection and tracking based on a security issue. In particular, researchers are mainly interested in autonomous system configuration, object identification, and multi-modal systems.

In [12], an architecture for implementing scene understanding algorithms in the visual surveillance domain has been presented. The agent paradigm is adopted to provide a framework in which inter-related and event-driven processes can be managed in order to achieve a high level description of events observed by multiple cameras. Each camera has associated an agent, which detects and tracks moving regions of interest. Each camera is calibrated in order to transform image co-ordinates into ground plane locations. By comparing properties, two agents can infer that they have the same referent, i.e. that two cameras are observing the same entity, and, as a consequence, merge identities. Agents store a hidden Markov model of learned activity patterns.

In [1], a video-based traffic surveillance multi-agent system, called Monitorix, is presented. Agent interactions are controlled by a BDI-like architecture. Agents communicate using FIPA-ACL messages with SL contents. Each agent defines a set of predicates, functions and actions that may be referred in the messages that it receives. Vehicles are tracked across cameras by the a suitable agent, using a traffic model whose parameters are continuously updated by learning algorithms. The classification of mobile objects uses competitive learning algorithms. The computation of typical trajectories uses statistical adaptation. The tracking of mobile objects, from one camera into the next, updates the parameters of its prediction model, using a combination of symbolic learning and genetic algorithms.

In [11], a coordinated video surveillance system, which can minimize the spatial limitation and can precisely extract the 3D position of objects, is presented. The proposed system uses an agent based system and also tracks the normalized object using active wide-baseline stereo method. The system is composed of two parts: multiple camera agents (CAs) and a support module (SM). Each CA treats image processing and camera controlling. SM are devoted to manage communication between CAs. The system extracts object positions independent of environment via the collaboration of CAs and a SM.

To our best knowledge, no agent-based solutions have been proposed in the literature to monitoring and signaling intrusions in marine reserves.

3 Scenario

In the summertime, in Sardinia and in its small archipelago there a lot of tourists that often sail in protected or forbidden areas and/or close to the coast. Monitoring such areas is quite complicated since the corresponding scenario requires to discriminate between authorized and not authorized boats.

Along the length of Sardinia coast, there are two-hundred tourist ports with about thirteen thousands places available for boats and several services for boat owners. Small ports typically have to monitor large areas in order to guarantee the access to authorized boats without suitable resources (e.g., radars). In this areas, staff operators have to directly patrol the surface in an uneconomic way.

A typical solution consists of using a radar system controlled by a central unit located ashore in a strategical position. Radar signals allow to detect the positions of the boats that sail in the controlled area. The main problem is that it is needed to distinguish among authorized and not authorized boats. In this paper, we present a novel down-market approach.

Being interested in monitoring and signaling intrusion in marine reserves, we decided to supply each authorized boats with suitable devices able to transmit (through the GSM technology) their position (through the GPS technology). In this way, the corresponding scenario encompasses two kinds of boats: authorized ones recognizable by the GPS+GSM devices, and not authorized ones. Both kinds of boats will be identified by a digital radar able to detect the position of all the mobile objects located in the protected area. Comparing the position sent by the boat and the one detected by the radar will easily help in identifying not authorized boats signaling intrusions to the staff operators. Furthermore, such approach may also allow to establish communication between the central server and the boats in order to send information about the weather or to provide assistance.

From a conceptual perspective, we can consider the problem of monitoring and signaling intrustions in marine reserves as an information retrieval task. In fact, a typical information retrieval task has to take into account the following issues:

i. how to deal with different information sources and to integrate new information sources without re-writing significant parts of it;
ii. how to suitably encode data in order to put into evidence the informative content useful to discriminate among categories;
iii. how to allow the user to specify her / his preferences;
iv. how to exploit the user feedback to improve the overall performance of the system.

As for the considered scenario, information sources are radar and GPS+GSM devices; the categories among discriminate with are authorized and not authorized boats; user preferences depend on the specific role of the user (e.g., system administrator or staff operators); and the feedback could be adopted by the system administrator to signal possible errors in the intrusion detection. The

above issues are typically strongly interdependent in information retrieval systems. To better concentrate on these aspects separately, we adopted radar and GPS+GSM devices together with a multiagent system, able to promote the decoupling among all aspects deemed relevant.

Moreover, resorting to a multiagent solution may help in dealing with the underlying scenario that is a distributed environment with limited resources, needs a light-weight task allocation, and has to support flexible and scalable requirements.

4 The Adopted Solution

The adopted multiagent system consists of a society of software agents, each embodying heterogeneous characteristics and responsibilities. Each agent is devoted to exhibit an intelligent behavior to provide a useful support to the final user in monitoring the boats. To this end, several kinds of agents have been devised: (i) agents aimed at extracting information from the information sources (i.e. the radar and the mobile devices); (ii) agents devoted to encode such information; (iii) cooperative agents that perform the role of domain experts and are aimed at following boats signaling intrusions; and (iv) interface agent through the user can interact with.

The proposed system has been devised by customizing to this specific task X.MAS, a generic multiagent architecture devised to implement information retrieval and filtering applications. For the sake of completeness, in this section we first summarize X.MAS (the interested reader may consult [2] for further information) and then illustrate its customization to monitor boats in marine reserves.

4.1 X.MAS

The X.MAS architecture encompasses four main levels: information, filter, task, and interface.

At the *information level*, agents are entrusted with extracting data from the information sources. Each information agent is associated to one information source, playing the role of wrapper.

At the *filter level*, agents are aimed at selecting information deemed relevant to the users, and at cooperating to prevent information from being overloaded and/or redundant. In general, two filtering strategies can be adopted: generic and personalized.

At the *task level*, agents arrange data according to users personal needs and preferences. Task agents are devoted to achieve user goals by cooperating together and adapting themselves to the changes of the underlying environment.

At the *interface level*, a suitable interface agent is associated with each different user interface. In fact, a user can generally interact with an application through several interfaces and devices (e.g., pc, pda, mobile phones, etc.).

X.MAS agents are JADE [3] agents that can (i) interact by exchanging FIPA-ACL messages, (ii) share a common ontology in accordance with the actual application, and (iii) exhibit a specific behavior according to their role. As for agent internals (see Figure 1), each agent encompasses a scheduler devoted to control the information flow between adjacent levels. Information and interface agents embody information sources and specific devices, respectively. Filter and task agents encompass an actuator that depends on the actual application. Middle agents contain a dispatcher aimed at handling interactions among requesters and providers.

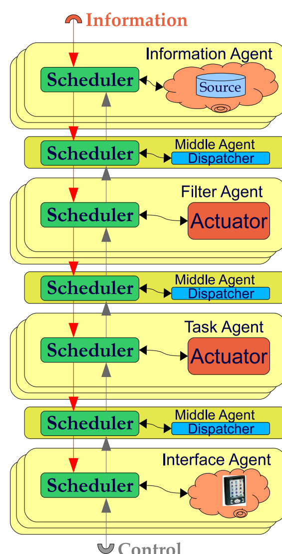


Fig. 1. Agent internals.

4.2 SEA.MAS: X.MAS for Monitoring Boats in Marine Reserves

Figure 2 sketches the macro-architecture of the implemented customization of X.MAS for monitoring boats in marine reserves, the corresponding system has been called SEA.MAS.

Information level. In this particular scenario, the information sources are the digital radar and the GPS+GSM devices. For each information source a suitable information agents will be devoted to embody the information provided by the corresponding source. To this end, we implemented a wrapper of the digital radar and a wrapper of the GPS+GSM device. Retrieved information becomes available to the other agents of the system by invoking the middle agent corresponding to the middle-span level "Information-Filter".

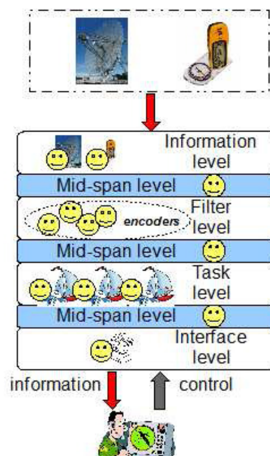


Fig. 2. SEA.MAS macro-architecture

Filter level. Filter agents are aimed at encoding the information extracted by the information agents in order to create events containing the position of the detected objects and its identify code, if available. Moreover, filter agents are devoted to avoid two kinds of redundancy: information detected more than once from the same device (caching) or by different devices (information overloading). Then, the middle agent corresponding to the middle-span level "Filter-Task" forwards the event to the corresponding task agent assuming the role of yellow pages if the identify code is available, or the role of broker otherwise. In case of the detected event does not correspond to an authorized boat, the middle agent creates the corresponding task agent able to handle the event.

Task level. To each boat corresponds a task agent, the underlying motivation being the necessity to centralize the knowledge regarding the boat position, its state and other possible further communication channels. As for the position, the events provided by the agents belonging to the upper level are taken into account also considering the past and are classified as follows: events belonging to anonymous detection systems and events belonging to known detection systems (i.e., with an identify code). As for the state, it depends on the identity code and/or on possible information provided by the corresponding boat. As for possible further communication channels, a bidirectional message exchange between the boats and the server could be also available (e.g., to provide weather information). The main tasks of the agents belonging to this architectural level are: (i) to follow a boat position during its navigation, also dealing with any temporary lack of signal; (ii) to promptly identify not authorized boats alerting the interface agents; and (iii) to handle messages coming from the interface level in order to notify the involved devices.

Interface level. A suitable interface agent allows users to interact with the system. Final users are the system administrator and staff operators. The

interface agent is also devoted to pass information from the user to the rest of the agents, for example to notify the interested agents about changes occurred in the environment.

Let us note that the corresponding system involves a number of agents that depends on the number of boats. In fact, whereas the number of information-, filter-, and interface-agents is fixed (i.e., two information-, one filter-, and one interface-agent), a task agent must be instantiated for each boat. This is not a problem for two main reasons: agent can be distributed on several nodes, and, typically, marine reserves can host about 20 boats at the same time as a maximum.

As for the capabilities that X.MAS agents can exhibit, in our opinion, in this particular scenario, the main ones are: (i) cooperation, (ii) mobility, (iii) personalization, (iv) adaptativity, and (v) deliberative capability.

Cooperation. Cooperation is the main requirement to be implemented in SEA.MAS agents. In particular, agents must cooperate to coordinate their actions in order to achieve their goals. In SEA.MAS cooperation may occur both horizontally and vertically. The former kind of communication supports cooperation among agents belonging to a specific level, whereas the latter supports the flow of information and/or control between adjacent levels through suitable middle-agents. Cooperation is implemented in accordance with the following modes: centralized composition, pipeline, and distributed composition (see Figure 3). Centralized compositions can be used for integrating different capabilities, so that the resulting behavior actually depends on the combination activity. Pipelines can be used to distribute information at different levels of abstraction, so that data can be increasingly refined and adapted to the user's needs. Distributed compositions can be used to model a cooperation among the involved components aimed at processing interlaced information. Communication among agents is performed by the FIPA-ACL support provided by JADE.

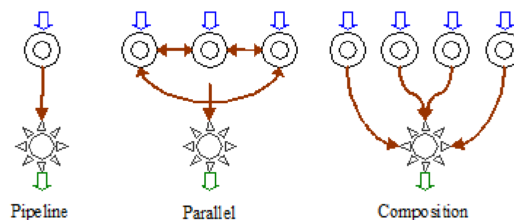


Fig. 3. Cooperation modes

Mobility. All the involved agents can be mobile, if needed. In fact, in case of a large number of agents (i.e., boats) this requirement becomes mandatory in order to handle with the computational complexity. Thus, mobility allow to divide the computation in several nodes. Let us recall that X.MAS agents are, in fact, JADE agents. So that, it is very easy to build mobile agents able to migrate

or copy themselves across multiple network hosts. In particular, JADE supports both intra-platform and inter-platform mobility, i.e., a JADE mobile agent can navigate across different agent containers and platforms. Mobile agents need to be location aware in order to decide when and where to move. Therefore, X.MAS provides a suitable ontology that holds the necessary concepts and actions.

Personalization. Personalization is provided to perform profiling at the interface level. In fact, the system is able to provide a different interface depending on the actual operator (e.g., system administrator and several kinds of staff operators). The information about the user profile is stored by agents belonging to the interface level. It is worth noting that, to exhibit personalization, filter and task agents may need information about the user profile. This flows up from the interface level to the other levels through the middle-span levels. In particular, agents belonging to mid-span levels (i.e., middle agents) take care of handling synchronization and avoiding potential inconsistencies. Moreover, the user behavior is tracked during the execution of the application to support explicit feedback, in order to improve her/his profile.

Adaptativity. Currently, SEA.MAS agents exhibit a very trivial adaptive capability. Task agents, in fact, are able to adapt their behavior in order to avoid to loose boats in case of signal absence (i.e., areas devoid of GSM signal). In a future release of the system, we are planning to implement multiagent learning strategies, such as evolutionary computation strategies, to allow task agents to self-adapt to further changes that may occur in the environment.

Deliberative capability. Currently, SEA.MAS agents do not exhibit deliberative capabilities. In a future release of the system, we are planning to implement reasoning algorithms that allow task agents to autonomously replan their actions if needed.

5 The Current Prototype

Figure 4 illustrates the components involved in the system.

First a simulator has been devised in order to test SEA.MAS. Such component simulates both the radar and the GPS+GSM devices. In particular, it simulates signals belonging to GPS and radar by randomizing temporal displacement and by adding detection errors. Furthermore, the simulator mimics the behavior of the boats by reproducing accelerations and flipping. The presence of GPS devices on the boats have been also simulated. To test the robustness of the approach, the prototype has been tested scaling up the number of boats from 20 to 100. Let us recall that, in the worst case (i.e., 100 boats) the system involves on the whole 104 agents. Performances put into evidence that, also considering 100 boats, the system is able to signal intrusions in a very few time.

To assess the capability of SEA.MAS in detecting intrusions, we perform experiments considering first a scenario with minus than 40 boats, and then a scenario with a number of boats varying from 40 to 100. In the former scenario, being boats enough spaced out, the filter agent is able to easily distinguish among authorized and not authorized boats and, consequently, task agents are correctly

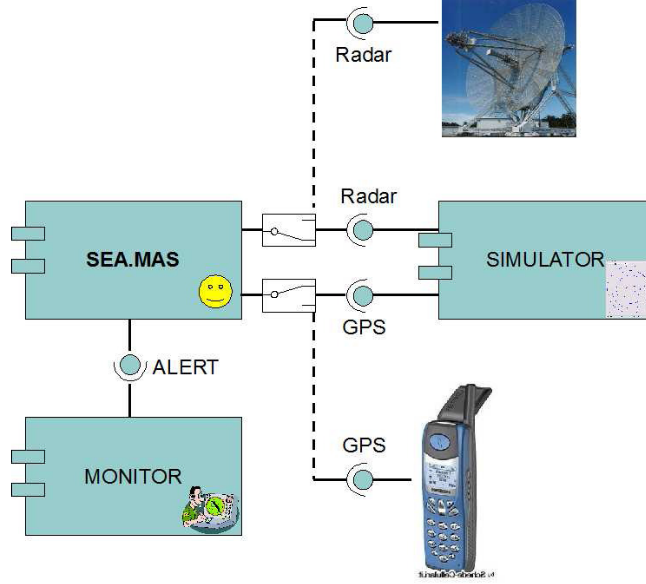


Fig. 4. Schema of the overall system

instantiated and able to signal the detected intrusions as soon as they occur. In the latter scenario, the signals provided by the boats could overlap, and, as a consequence, the filter agent could not correctly distinguish among authorized and not authorized boats. In this case, some task agent could be associated to not authorized boats even if they actually correspond to an authorized one, and vice versa. To measure such error, we calculate the confusion matrix as follows: true positives (TP) are the not authorized boats recognized as intrusions; true negatives (TN) are the authorized boats recognized as not intrusions; false positives (FP) are authorized boats recognized as intrusions; and false negatives (FN) are not authorized boats recognized as not intrusions. Starting from those values we then calculated accuracy (α), precision (π), and recall (ρ), as follows:

$$\alpha = \frac{TP + TN}{TP + TN + FP + FN} \quad (1)$$

$$\pi = \frac{TP}{TP + FP} \quad (2)$$

$$\rho = \frac{TP}{TP + FN} \quad (3)$$

Experiments showed that, on the average, the system accuracy is 95%. As for precision and recall, on the one hand the system showed the same propensity to make mistake in FP and FN, on the other hand the overall error is quite low. On the average, precision and recall are 93%. Furthermore, for both scenarios we test the system varying the percentage of positive (not authorized boats, i.e.,

intrusions) and negative (authorized boats, i.e., not intrusions) examples. In both cases, the system is not influenced by those variations in the inputs, meaning that the behavior of the system is independent from the kind of involved boats.

Once SEA.MAS has been tested, the simulator has been switched off and the system has been used to monitor boats in a marine reserve located in the north Sardinia. The GPS- and radar-signals are retrieved by suitable information agents that are able to extract the actual position according to GPS and NMEA standards. As for the interface agent, it represents in different colors the different states corresponding to the boats: authorized, not authorized, not-detected, under verification. In case of intrusions, an acoustic sound is generated together with the position of the not authorized boats. Such signal is then sent to the boats devoted to reach the not authorized boat. The maximum number of boats in the selected marine reserve was 20. Experiments performed on the real scenario showed results comparable with the ones performed during the simulation.

Summarizing, results show that adopting the proposed approach allows system administrator and staff operators to easily identify intrusions. Results are quite interesting also considering that the system is down-market.

Finally, SEA.MAS has been also used to monitor a tourist port. In this application, the sistem is able to detect arriving boats that have booked a place in order to provide them all the required facilities as soon as they arrive. In Figure 5 the interface of this application is showed.

Fig. 5. The application to monitor a tourist port

6 Conclusions and Future Work

In this paper, SEA.MAS, a multiagent system for monitoring boats in marine reserves, has been presented. The system has been built upon X.MAS, a generic multiagent architecture devoted to support the implementation of information retrieval applications managing information among different and heterogeneous sources. The system has been used to monitor boats in a marine reserve located

in the north Sardinia. Results show that adopting the proposed approach allows system administrator and staff operators to easily identify intrusions.

As for the future work, we are implementing a new release of the system in which further information about weather and/or points of interest will be also provided. Furthermore, task agents will also exhibit adaptive and deliberative capabilities. Finally, more sophisticated user profiling techniques for the interface agent implementation are currently under study.

7 Acknowledgments

This work has been supported by Regione Autonoma della Sardegna, under the project “A Multiagent System for Monitoring Intrusions in Marine Reserves” (POR Sardegna 2000/2006, Asse 3 - Misura 3.13). The prototype has been developed and deployed together with SETI snc and ICHNOWARE sas.

References

1. B. Abreu, L. Botelho, A. Cavallaro, D. Douxchamps, T. Ebrahimi, P. Figueiredo, B. Macq, B. Mory, L. Nunes, J. Orri, M. Trigueiros, and A. Violante. Video-based multi-agent traffic surveillance system. *Intelligent Vehicles Symposium, 2000. IV 2000. Proceedings of the IEEE*, pages 457–462, 2000.
2. A. Addis, G. Armano, and E. Vargiu. From a generic multiagent architecture to multiagent information retrieval systems. In *AT2AI-6, Sixth International Workshop, From Agent Theory to Agent Implementation*, pages 3–9, 2008.
3. F. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)*. John Wiley and Sons, 2007.
4. F. Bergenti, G. Rimassa, A. Poggi, and P. Turci. Middleware and programming support for agent systems. In *Proceedings of the 2nd International Symposium from Agent Theory to Agent Implementation*, pages 617–622, 2002.
5. T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American Magazine*, 2001.
6. A. Bieszczad, T. White, and B. Pagurek. Mobile agents for network management. *IEEE Communications Surveys*, 1(1):2–9, 1998.
7. L. Bodrozic, D. Stipanicev, and M. Stula. Agent based data collecting in a forest fire monitoring system. *Software in Telecommunications and Computer Networks, 2006. SoftCOM 2006. International Conference on*, pages 326–330, 29 2006-Oct. 1 2006.
8. S. Bussmann, N. R. Jennings, and M. Wooldridge. On the identification of agents in the design of production control systems. In *Agent-Oriented Software Engineering, LNCS*, pages 141–162. Springer-Verlag, 2001.
9. I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann: San Francisco (CA), 1999.
10. J. O. Kephart. Software agents and the route to the information economy. *Proc. Natl. Acad. Sci.*, 99(3), 2002.
11. N. Kim, I. jae Kim, and H. gon Kim. Video surveillance using dynamic configuration of mutiple active cameras. *Image Processing, 2006 IEEE International Conference on*, pages 1761–1764, Oct. 2006.

12. J. Orwell, S. Massey, P. Remagnino, D. Greenhill, and G. A. Jones. A multi-agent framework for visual surveillance. In *ICIAP '99: Proceedings of the 10th International Conference on Image Analysis and Processing*, page 1104, Washington, DC, USA, 1999. IEEE Computer Society.
13. M. Pechoucek and V. Marik. Industrial deployment of multi-agent technologies: review and selected case studies. *Journal Autonomous Agents and Multi-Agent Systems*, 17(3):397–431, 2008.
14. R. Quitadamo and F. Zambonelli. Autonomic communication services: a new challenge for software agents. *Journal Autonomous Agents and Multi-Agent Systems*, 17(3):457–475, 2008.
15. W. Shen and D. H. Norrie. Agent-based systems for intelligent manufacturing: a state-of-the-art survey. *Knowledge and Information Systems, an International Journal*, 1(2), 1999.
16. D. Stipanicev, M. Stula, and L. Bodrozic. Multiagent based greenhouse telecontrol system as a tool for distance experimentation. *Industrial Electronics, 2005. ISIE 2005. Proceedings of the IEEE International Symposium on*, 4:1691–1696, 20-23, 2005.
17. F. Zambonelli and A. Omicini. Challenges and research directions in agent-oriented software engineering. *Journal of Autonomous Agents and Multiagent Systems*, 9:253–283, 2004.

A Middleware for modeling Organizations and Roles in Jade

Matteo Baldoni[1], Guido Boella[1], Valerio Genovese[1],
Andrea Mugnaini[1], Roberto Grenna[1] and Leendert van der Torre[2]

[1]Dipartimento di Informatica. Università di Torino - IT.
E-mail: {baldoni,guido,grenna}@di.unito.it; {mugnaini81,valerio.click}@gmail.com
[2]Computer Science and Communications, University of Luxembourg, Luxembourg
E-mail: leon.vandertorre@uni.lu

Abstract. Organizations and roles are often seen as mental constructs, good to be used during the design phase in Multi Agent Systems, but they have also been considered as first class citizens in MAS, when objective coordination is needed. Roles facilitate the coordination of agents inside an organization, and they give new abilities in the context of organizations, called powers, to the agents which satisfy the requirements necessary to play them. No general purpose programming languages for multiagent systems offer primitives to program organizations and roles as instances existing at runtime, so, in this paper, we propose our extension of the Jade framework, with primitives to program in Java organizations structured in roles, and to enable agents to play roles in organizations. We provide classes and protocols which enable an agent to enact a new role in an organization, to interact with the role by invoking the execution of powers, and to receive new goals to be fulfilled. Since roles and organizations can be on a different platform with respect to the role players, the communication with them happens via protocols. Since they can have complex behaviours, they are implemented by extending the Jade agent class. Our aim is to give to programmers a middle tier, built on the Jade platform, useful to solve with minimal implementative effort many coordination problems, and to offer a first, implicit, management of norms and sanctions.

1 Introduction

Roles facilitate the coordination of agents inside an organization, giving new abilities in the context of organizations, called powers, to the agents which satisfy the requirements necessary to play them. Organizations and roles are often seen as mental constructs, good to be used during the design phase in MAS, but they have also been considered as first class citizens in multiagent systems [8], when objective coordination is needed. No general purpose programming languages for multiagent systems offer primitives to program organizations and roles as instances existing at runtime, yet.

So, this paper answers the following research questions:

- How to introduce organizations and roles in a general purpose framework for programming multiagent systems?
- Which are the primitives to be added for programming organizations and roles?

- How it is possible to restructure roles during runtime?

Another subquestion could be the following: what does it bring to program roles and organisations as instances?

As methodology, we build our proposal as an extension of the Jade multiagent system framework, with primitives to program, in Java, organizations structured in roles, for enabling agents to play roles in organizations. As ontological model of organizations and roles we select [6] which merges two different and complementary views or roles, providing an high level logical specification.

To pass from the logical specification to the design and implementation of a framework for programming multiagent systems, we provide classes and protocols which enable an agent to enact a new role in an organization, to interact with the role by invoking the execution of powers (as intended, in OO programming, in [7], and shortly explained in Section 2.4), and to receive new goals to be fulfilled. Since roles and organizations can be on a different platform with respect to the role players, the communication with them happens via protocols. Since they can have complex behaviours, they are implemented by extending the Jade agent class. Our aim is to give to programmers a middle tier, built on the Jade platform, useful to solve with minimal implementative effort coordination problems.

We test our proposal on a possible scenario, highlighting the features of our model.

In this paper we do not consider the possibility to have BDI agents, even if both the ontological model (see [7]) and the Jade framework allow such extension.

The paper is organized as follows. First, in Section 2, we summarize the model of organizations and roles we take inspiration from, and we give a short description of our concept of "power". In Section 3, we describe an example of a typical MAS situation in the real life; in Section 4 we describe how our model is realized introducing new packages in Jade; in Section 5 we discuss a possible powerJade solution to a practical problem (the manager-bidder one), and Section 6 will finsh this paper with related work and conclusions.

2 The model of organizations and roles

Since we speak about organizations and roles, we need to refer to a formalized ontological model, in order to avoid ad hoc solutions imposed by the Jade framework, and to make understandable to programmers how to use the primitives. In the following subsections we shortly show two different (but complementary) views about roles (see [7] and [9]), and we introduce a unified model starting from these, and define a well-founded metamodel. Then, we explain our concept of "power".

2.1 The Ontological Model for the Organization

In [7] an ontological analysis shows the following properties for roles:

- *Foundation*: a role instance has always to be associated to an instance of the organization to which it belongs, and to an instance of the player of the role too;

- *Definitional dependence*: the role definition depends from the one of the organization to which it belongs;
- *Institutional powers*: the operations defined into the role can access the state of the organization, and of the other roles of the organization too;
- *Prerequisites*: to play a role, it is necessary to satisfy some requisites, that means that the player has to be able to do actions which can be used in the role's operations execution.

Also the model of [7] is focused on the definition of the structure of organizations, given their ontological status, which is only partly different from the one of agents or objects. On the one hand, roles do not exist as independent entities, since they are linked to organizations. Thus, they are not components like objects. Moreover, organizations and roles are not autonomous and act via role players. On the other hand, organizations and roles are description of complex behaviours: in the real world, organizations are considered legal entities, so they can even act like agents, albeit via their representative playing roles. So, they share some properties with agents, and, in some respects, can be modelled using similar primitives.

2.2 The Model for the Role Dynamics

[9]'s model focus on role dynamics, rather than on their structure; four operations to deal with role dynamics are defined: *enact* and *deact*, which mean that an agent starts and finishes to occupy (play) a role in a system, and *activate* and *deactivate*, which means respectively that an agent starts executing actions (operations) belonging to the role and suspends their execution. Although it is possible to have an agent with multiple roles enacted simultaneously, only one role can be *active* at the same time: when an agent performs a power, he is playing only one role in that moment.

2.3 The Unified Model

Using the distinction of Omicini [19], we use the model presented in [7] as an objective coordination mechanism, in a similar way, for example, artifacts do: organizations are first class entities of the MAS rather than a mental construction which agents use to coordinate themselves. However, this model leaves unspecified how, given a role, its player will behave. So, we merge it with [9]'s model, to solve the problem of formally defining the dynamics of roles, by identifying the actions that can be done in a *open system*, such that agents can enter and leave. Organizations are not simple mental constructions, roles are not only abstractions used at design time, and players are not isolated agents: they are all agents interacting the one with the others. A logical specification of this integrated model can be found in [6].

2.4 "Powers" in our view

We knows that roles work as "interfaces" between organizations and agents, and they give so called "powers" to agents. A power can extend agents abilities, allowing them to operate inside the organization and inside the state of other roles. An example of such

powers, called "institutional powers" in [17], is the signature of a director which counts as the commitment of the entire institution.

The powers added to the players, by mean of the roles, can be different for each role and, thus, represent different affordances offered by the organization to other agents to interact with it [4].

Powers are invoked by players on their roles, but they are executed by the roles, since they own both state and behaviour.

3 An example of MAS in real life

We will start with a real-life example, in order to explain a common situation that could be modeled with a Multi Agent System application. The scenario we want to consider involves two organizations: a bank, and a software house. Bob has been engaged as a programmer in a software house. The software house management imposes to him the owning of a bank account, in order to directly deposit his salary on it. Bob goes to the bank, where the employee, George, gives him some templates to fill. Once that Bob finished compiling the modules, George inputs the data on the terminal, creating the new account, which needs to be activated. George forwards the activation request to his director, Bill, who is the only able to activate an account in all the bank. Once that the account will be activated, Bob will be a new bank customer.

Years later, become a project manager, Bob decides to buy a little house. He has to obtain a loan, and the bank director informs him that for calling a loan, his wage packet is needed. Bob calls to the management of the software house for his wage packet, and bring it to Bill. After some days (and other templates filled), the bank gives the loan to Bob, who can finally buy his new house.

Each organization *offers* some roles, which have to be *played* by some agents, called, for this reason, *players*. In the bank, Bob plays the *customer* role, while George plays the *employee* one, and Bill the *director* one. Since Bob interacts with both the organizations, he has to play a role also inside the software house: he enters as a *programmer*, but after some years he changes it, becoming a *project manager*. As a bank customer, Bob has some *powers*: to call for an account, to transfer money on it, to request for a loan. George, being a simple employee, has the power to create Bob's account, but the account activation has to be done by Bill, the director. The call for activation is done by mean of a specific George's call to Bill, for the execution of a *responsibility*. Also in the case of the loan request, the director has to manage the situation, maybe examining Bob's account, and calling him for his wage packet. Another Bob's power is to call for his wage packet into the software house. Speaking about personal capabilities, we can imagine that Bill, in order to access to the bank procedures for which he is enabled, must fill a login page with his ID and password; the same happens for George too, and for Bob, in the moment in which he access to his account using Internet. Bob, however, has also another capability, that is *requested* when he plays the programmer role (but the same happens for the project manager one): to give his login name and password for entering the enterprise IT system. Finally, the director is required to have more complex capabilities, like evaluating the solvency of a client requesting a loan.

4 PowerJade

The main idea of our work is to offer to agents programmers a complete middle tier with the primitives for implementing organizations, roles, and players in Jade (see Figure 1). We called this middleware *powerJade*, remembering the importance of powers in the interaction between roles and organizations. The powerJade conceptual model is inspired to *open systems*: participants can enter in and leave from the system whenever they want. For granting this condition, and for managing the (possible) continuous operations for enacting, activating, deactivating, and deacting roles (in an asynchronous and dynamic way), many protocols have been realized. Another starting point has been the re-use of the software structure already implemented in powerJava [5], based on an intensive use of so-called *inner classes*.

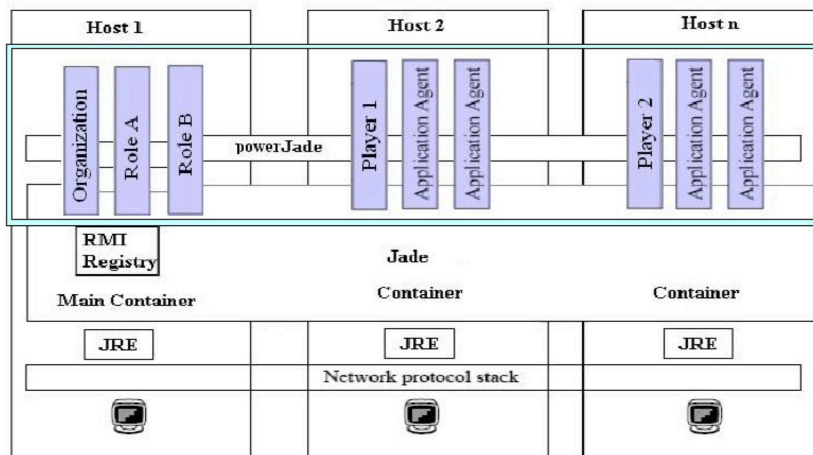


Fig. 1. The Jade architecture and the powerJade middle tier.

In order to give an implementation based on the conceptual model we discussed in Section 2.3, not only the three subclasses of the Jade `Agent` class (`Organization`, `Role`, and `Player`) have been realized (they will be described in Sections 4.1, 4.2, 4.3), but also classes for other central concepts, like `Power`, and `Requirement` were implemented (and showed in Sections 4.2, 4.3). For representing the dynamics of the roles, we implemented also all the needed communication protocols, that will be described in Section 4.4.

`Organization`, `Role`, and `Player` have similar structures: they contain a finite state machine behaviour instance which manages the interaction at the level of the new middle tier by means of suitable protocols for communication.

To implement each protocol in Jade two further FSMBehaviour are necessary, each one dealing the part of the protocol of the two interactants; for example, the enactment

protocol between the organization and the player requires two FSMBehaviours, one in the organization and one in the player.

4.1 The `Organization` Class

The `Organization` class is structured as in Figure 2. The `OrgManagerBehaviour` is a finite state machine behaviour created inside the `setup()` method of `Organization`. It operates in parallel with other behaviours created by the programmer of the organization, and allows the organization to interact via the middle tier. Its task is to manage the enact and deact requests done by the players. At each iteration, the `OrgManagerBehaviour` looks for any message having the *ORGANIZATION_PROTOCOL* and the performative *ACLMessage.Request*. `EnactProtocolOrganization` and `DeactProtocolOrganization` are the counterpart of the respective protocols inside the players which realize the interaction between organizations and players: instances of these two classes are created by the `OrgManagerBehaviour` when needed.

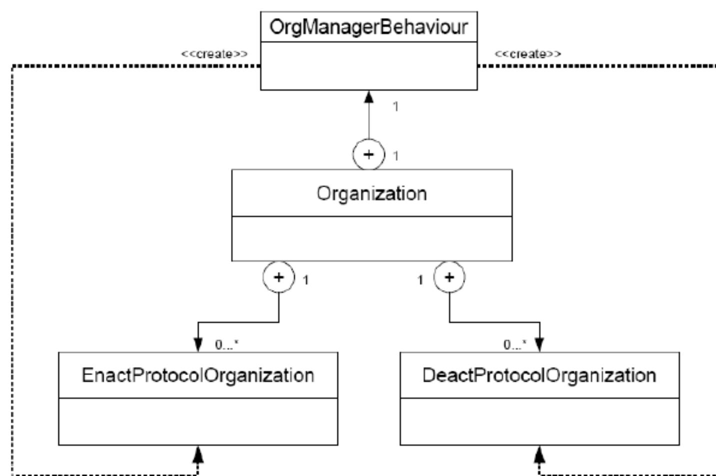


Fig. 2. The Organization diagram.

When the `OrgManagerBehaviour` recognize a message to manage, it extracts the sender's AID, and the type of request required. In case of an *Enact* request (and whether all the controls described on Subsection 4.4 about the *Enact* protocol succeeded), a new instance of `EnactProtocolOrganization` behaviour is created, and added to the queue of behaviours to be executed. The same happens (with a new instance of the `DeactProtocolOrganization` behaviour) if a *Deact* request has been done, while if the controls related to the requested protocol will not succeed, the iteration terminate, and the `OrgManagerBehaviour` takes again its cycle. In the behavioural part of this class, programmers can add a "normative" control on the players'

good intentions, and managing the possibility of discovering lies before enacting the role, or immediately after having enact it (and before w.r.t. its activation). Primitives implementing these controls are ongoing work.

4.2 The `Role` Class

As described in [3], the `Role` class is an `Agent` subclass, but also an `Organization` *inner class*. Using this solution, each role can access to the internal state of the organization, and to the internal state of other roles too. Like the `Organization` class has the `OrgManagerBehaviour`, the `Role` has the `RoleManagerBehaviour`, a finite state machine behaviour created inside the `setup()` method of `Role`. Its task is to manage the commands (messages) coming from the player: a power invocation, an *Activate*, or a *Deactivate*.

Inside the role, an instance of the `PowerManager` class is present. The `PowerManager` is a `FSMBehaviour` subclass, and it has the whole list of the powers of the role (linked as states of the FSM). It is composed as follows:

- a first state, the `ManagerPowerState`, that must understand which power has been invoked;
- a final state, the `ResultManager`, that has to give the power result to its caller;
- a self-created and linked state for each power implemented by the role programmer.

All the transitions between states are added at run-time to the FSM, respecting the code written by the programmer.

The Powers Powers are a fundamental part of our middleware. They can be invoked by a player on the active role in the particular moment of the invocation, and they represent the possibility of action for that role inside the organization. For coherence with the Jade framework and to exploit the scheduling facility, powers are implemented as behaviours, getting also advantage of their more declarative character with respect to methods.

Some times, a power execution needs some requirements to be completed; this is a sort of remote method call dealt by our middleware, since requirements are player's actions. In our example, George, as bank employee, has the *power* of creating a bank account for a customer; to exercise this power, George as player has to input his credentials: the login and the password.

The problem to be solved is that players' requirement invocation must be transparent to the role programmer, who should be relieved from dealing the message exchange with the player.

We modeled the class `Power` as a `FSMBehaviour` subclass, where the complete finite state machine is automatically constructed from a declarative specification containing the component behaviours to be executed by the role and the name of the requirements to be executed by the player; in this way, we can manage the request for any requirement as a particular state of the FSM. When a requirement is required, a `RequestRequirementState` (that is another subclass of `FSMBehaviour`) is

added automatically in the correct point invoking the required requirement by means of a protocol: the programmer has only to specify the requirement name.

The complexity of this kind of interaction is shown in Figure 3. The great balloon indicating one of the powers for that particular role contains the final state machine obtained writing the following code:

```
addState(new myState1("S1", "R1", "E1"));
addState(new myState2("S2"));
```

where *S1* and *S2* are names of possibly complex behaviours implemented by the role programmer which will be instanced and added to the finite state machine representing the power, *R1* is the name requested requirement, and *E1* is a behaviour representing the error management state. Analyzing the structure of the power, we can see that the execution of the first state *S1* is followed by a macro-state (that is a `FSMBehaviour`), managing the request for a requirement, automatically created by the `addState()` method. This state will send to the player the request for the needed requirement, also managing the possible parameters, waiting for the answer. Whether the answer is positive, the transition to the following state of the power is done (or to the `ResultManager`, if needed); otherwise, the error can be managed (if possible), or the power is aborted. The `ErrorManager` is a particular state that allows to manage all the possible kinds of error, also the case in which a player lied about its requirements).

Error management is done via the middle tier. We can individualize two kinds of possible errors: (i) the *accidental* ones, and (ii) the *voluntary* ones. Typical cases of the (i) are the "practical" problems (i.e. network too busy and timeout expired), or the ones linked to a player bad working (also, a programming problem); those indicated as (ii) are closely linked to an incorrect behaviour of the player, like the case in which an agent lied on its requirements during an enact protocol. The latter case of error managing allows to the organization and roles programmer a fist, rough, implicit, normative and sanctionative mechanism: if the player, for any reason, shows a lack of requirements, it could be obliged to the deact protocol w.r.t. that particular role, or it can be "marked" with a negative score, that could mean a lower trust level exercised from the organization to it.

An advantage given by using a declarative mechanism like behaviours for modelling powers is that new powers can be dynamically added or removed from the role. It is sufficient to add or remove transactions linking the power to the `ManagerPowerState` which is a FSMBehaviour too.

This mechanism can be used to model both dynamics of roles in organizational change or access restrictions. In the former case we can model situations like the power of the director to add to the employee the power of giving loans. In the latter case, we can model security restriction by removing powers from roles, so to avoid the situation where first a power is invoked and then aborted after controlling an access control list.

4.3 The `Player` Class

Analogously to `Organization` and `Role`, also the `Player` class is an `Agent` subclass. Like in the other two cases, we have a `PlayerManagerBehaviour`, a FSM-

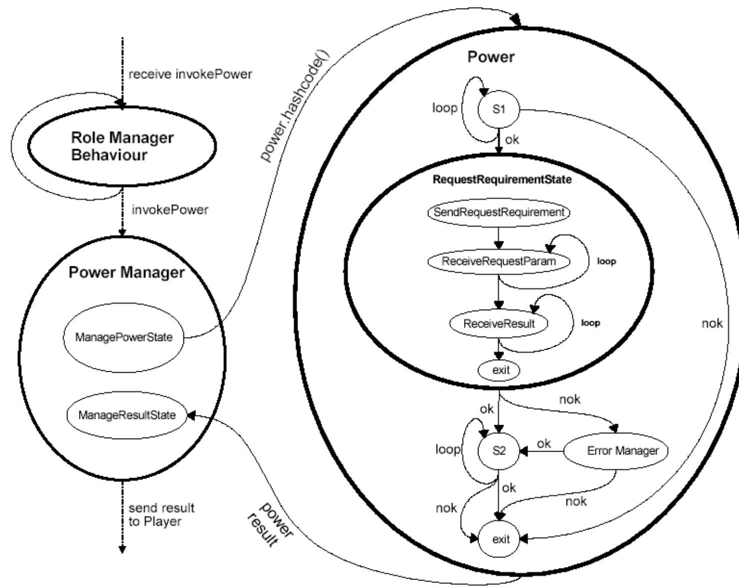


Fig. 3. Power management.

Behaviour managing all the possible messages that the player can receive. The player is the only agent totally autonomous. It contains other behaviours created by the agent programmer which are scheduled in parallel with the manager behaviour and it can obviously also interact with other agents, not involved in any organization (since the communication protocol existing in Jade always continues working), but it's constrained to interact with any kind of organization using a role offered by the organization itself. In case of a communication with another agent inside the organization, it can be done only via roles. Any other activity, communication, or action that both the agents could do without passing through their roles will not have effect on the internal state of the organization at all. Only the player can use all the four protocols described in Subsection 2.2: *Enact* and *Deact* with the organization, *Activate* and *Deactivate* with the role. While the role has to manage powers, the player deals with requirements: this is done by a `RequirementManager`.

The `Player` class offers some methods. They can be used in programming the other behaviours of the agent when it is necessary to make change to the state of role playing or to invoke powers. We assume invocations of powers to be asynchronous via the `invokePower` method from any behaviour implemented by the programmer. The call informs the `PlayerManagerBehaviour` which starts the interaction with the role and returns a call id which is used to receive the correct return value in the same behaviour if necessary. It is left to the programmer how to manage the necessity of blocking of the behaviour till an answer is returned, with the usual *block* instruction of JADE. This solution is coherent with the standard message exchange of JADE and allows to avoid using more sophisticated behaviours based on threads. The behaviour can

then consult the `PlayerManagerBehaviour` to get the return value of the power if it is available.

The player, once having invoked a power, stays waiting, i.e., for messages o requests from the active role. When the role needs for some requirements, the `PlayerManagerBehaviour` passes the control to the `RequirementManager`, which execute all the tasks which are needed.

It's important to notice that a player can always grow w.r.t. its capabilities/requirements.

A player can know organizations and roles on the platform by using the *Yellow Pages* mechanism, that in a basic JADE feature.

The Requirements Requirements are, for a player, a subset of the behaviours representing its capabilities, and, in some sense, the plans for achieve the personal goals of the agent. Playing a role, an agent can achieve more goals (i.e., the goals achievable invoking a power), but, in a general case, the execution of one or more requirements can be needed during the invocation of a power. Referring to our bank example, George can achieve many goals dealing with its employee role (i.e., create a new account), but to do it, it's necessary for him to log in inside the bank IT system. Seen as a requirement, its log in capability denote his "attitude", his "possibility" of playing his employee role.

During the enact protocol, the organization sends (see Section 4.4) to the agent wanting to play one of its roles, the list of requirements to be fulfilled. As we said, the candidate player could lie, entering in the role in a not honest way. The organization and role programmer, however, has all the possibility to check the truth of the candidate player's answer before it begins to play the role, not enacting it, or deacting immediately after the enact. Also this kind of choice has been done to grant the highest freedom degree

4.4 Communication Protocols

In this Section, an example of a complex communication between a player, an organization, and a role is shown. We have to make some preliminary considerations, about communication. Each protocol is split in two, specular, but complementary behaviours, one for each actor. In fact, if we consider a communication, two "roles" can be seen: an initiator, which is the object sending the first message, and a responder, which never can begin a communication. For example, when a player wants to play a role inside an organization, an `EnactProtocolPlayer` instance is created. The player is the initiator, and a request for a role is done from its new behaviour to the `OrgManagerBehaviour`, which instantiates an `EnactProtocolOrganization` behaviour. This behaviour will manage the request, sending to the `EnactProtocolPlayer` an `Inform` containing the list of the requirement needed to play the requested role.

The `EnactProtocolPlayer` evaluates the list, answering to the organization part whether it agrees (notice that the player programmer could implement a behaviour that always answers in a positive way, that sounds like a lie). Only after receiving the agreement, the `EnactProtocolOrganization` creates a `RoleManager` instance, and sends the AID of the role just created to the player. The protocol ends with the update by the player of its internal state.

Since the instance of a role, once created, is not yet activated, when the player wants to "use" a role, has to activate it. Only one role at a time is active, while the others, for which the agent finished successfully the enactment protocol, are deactivated. The activation protocol moves from the player to the role instance. The player creates an `ActivateProtocolPlayer`, which sends a message to the role, calling for the activation. This message produces a change into the internal state of the role, which answers with an `inform` telling its agreement.

Once the role has been activated, the player can proceed with a power invocation. As we discussed in [3], this is not the only way in which player and role instance can communicate. We consider it, since it can require a complex interaction, beginning from the `invoke` done by the player on a power of the role. As we shown in Subsection 4.2, the power management can involve the request to the player for the execution of one or more requirements. In this case, the role sends a `request` with the list of requirements to be fulfilled. The player, since autonomous, can evaluate the opportunity to execute the requirement(s), and take the result(s) to the role (using an `inform`, waiting for the execution of the power and for receiving the `inform` with the result. A particular case, not visible in Figure 4, is the one in which the player, for any reason, does not execute the required requirements. This "bad" interaction will finish with an automatic deactment of the role.

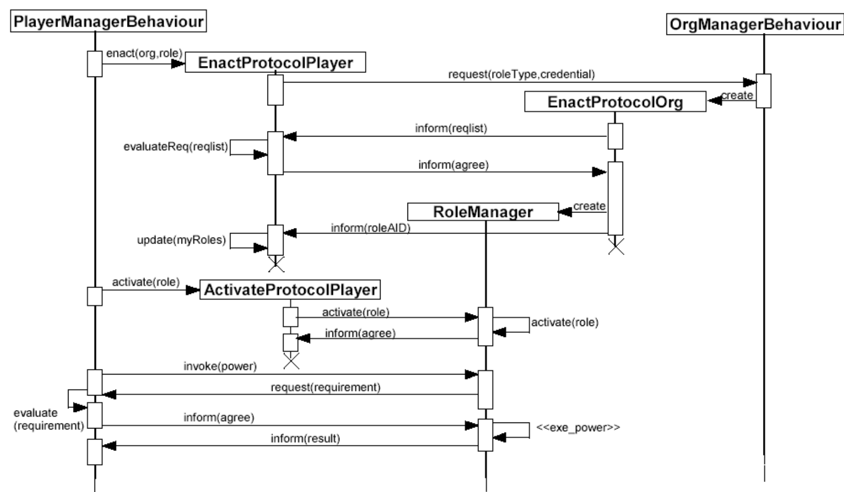


Fig. 4. The Sequence Diagram for a complex communication.

5 The CNP scenario in powerJade

In Section 3, we discussed the bank example, trying to focus on roles' powers, players' requirements, responsibility calls, and all that has a place in our middleware. In this

Section, we want to show a more technical example: the CNP one, or manager-bidder problem. In Figure 5, a little part of the interaction between the player for the manager role and its role is shown. Let's consider an agent doing one of its behaviours. In a particular moment, a task has to be executed, but the agent knows that it cannot execute it, since this job is not achievable with its capabilities. The only solution is to find someone able to execute the task, possibly paying the least is possible. The agent has no knowledge about the ContractNet Protocol, but it knows that there is an organization that offers the CNP by mean of its roles. The (candidate) player contacts the organization, starting the enact protocol for the role of manager in the CNP `M_CNP`. The organization sends the list of requirements to be fullfilled, composed by the "task" requirement (that is the ability to send a task for a call for proposal operation), and the "evaluate" task (that is the ability to evaluate the various bidders' proposals, choosing the best one). The candidate player owns the requirements, so the role is created. When the player come to execute once again the behaviour containing the not executable task, an `invokePower()` is executed, calling for the power with name *CNP* (the bold arc with number 1 in Figure 5). The role begins the power execution (managed by the `PowerManager`, after the `RoleManager` has passed to it the control). The first state for the power is the request for a requirement: for starting a call for proposal, the task to be delegated must be specified by the player. The `RequestRequirementState` sends a request for requirement to the `PlayerManager` (the bold arc with number 2 in Figure 5), that passes the control to the `RequirementManager`. The correct requirement is executed (the state which entering arc is labeled "task"), and the result is sent back to the `RequestRequirementState` (the bold arc with number 3). The power execution goes on, arriving to the `SEND_CFP` state, that provides the call for proposal to any bidder known inside the organization (bold arc with label 4, we assume that some agents already enacted the bidder role), going directly to add the opportune behaviour to the `PowerManager` of the `B_CNP` instances found. The bidder roles will send messages back to the manager roles, after requesting to their players the requirement to specify or not a price for the task to be delegated.

The complicated interaction between players and their roles, and between role and role, is executed *without* that players have to know the CNP dynamics, since all the complexity has been introduced in the roles. For the player playing the manager role, and for the ones playing the bidder role, the organization is a kind of black box; roles are the "wizards" managing the communication logics, and opportunely calling operations to be done by the players (that are absolutely *autonomous*: they are the only agents able to take decisions.

6 Related work and conclusions

On organizations and roles representations, many models have been proposed [12], applications modeling organizations or institutions [19], software engineering methods using organizational concepts like roles [25]. Several agent programming languages (among which 3APL [24]) have been developed, but few of them have been endowed with primitives for modeling organizations and roles as first class entities. Exceptions can be found in MetateM [11] (which is BDI oriented, is based on the notion of group,

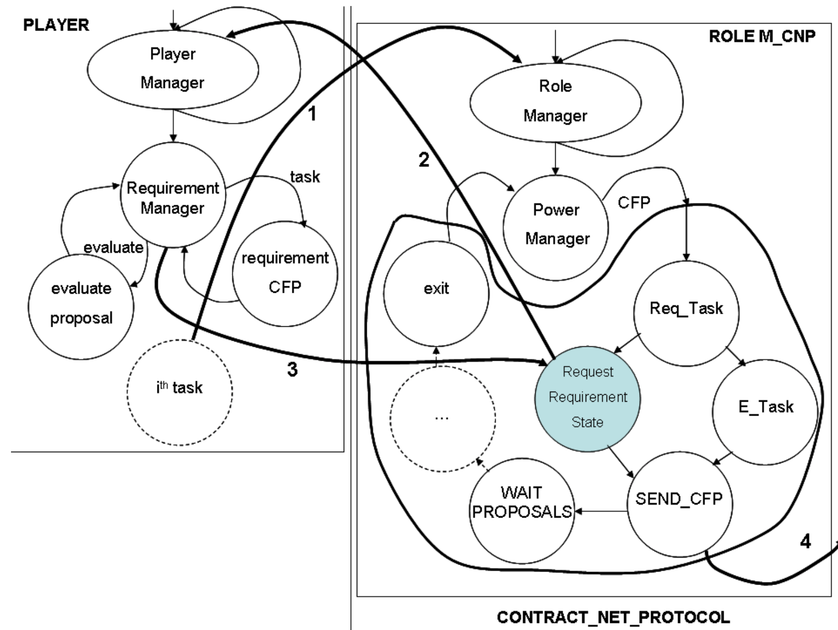


Fig. 5. Part of the solution for the CNP example. We can notice three interactions between different actors: (1) is from a player's behaviour to the active role; (2) is from a role's power to the player; (3) is from a the player to the role, communicating the requirement result; (4) is from a role's power to another role.

and it is not a general purpose language), J-MOISE+ [15] (which is more oriented to programming *how* agents play roles in organizations), and the Normative Multi-Agent Programming Language in [22] (which is more oriented to model the institutional structure composed by obligations, more than the organizational structure composed by roles). Considering frameworks for modelling organizations like SMoise+ [16] and MadKit [13], can be noticed limited possibilities to program organizations.

Regarding the analysis of organizations, in [23] can be found what is called the perspective of computational organization theory and artificial intelligence, in which organizations are basically described at the role, and group, composed of roles, levels. Under this perspective, works such as GAIA [25] (which is a model for designing MAS, more than a framework) and the already cited (with extensions) MOISE [14] can be found, while other models, such as ISLANDER [10], define organizations as electronic institutions, in terms of norms and rules.

With respect to organizational structures, Holonic MAS [21] present particular pyramidal organizations in which agents of a layer (under the same coordinator, also known as the holon's *head*) are able to communicate and to negotiate directly between them [1]. Roles and groups can express quite naturally Holonic structures, under the previously described perspective.

Looking at agent platforms, there are two other—other than JADE—which can be considered relevant in this context. First, JACK Intelligent Agents [2] supports organizational structures through its Team Mode, where goals can be delegated to team member in order to achieve the team goals. JADEX [20] presents another interesting platform for the implementation of organizations, even if it does not currently have organizational structures.

[18] make a very similar proposal to powerJade. However, it does not propose a middle tier supported by a set of managers and behaviours making all the communication transparent to agent programmers. It presents a simpler approach that relies mostly on the extension of agents through behaviours and represents Roles as components on an ontology, while our approach presents a slightly more complex approach, in which roles are implemented as agents that provide further decoupling by brokering between organizations and players, and provides a state machine that permits precise monitoring of the state of the roles.

In this paper we introduce organizations and roles as new classes in the Jade framework which are supported by a middle tier offering to agents the possibility to enact roles, invoke powers and to coordinate inside an organization.

The framework is based on a set of FSMBehaviours which realize the middle tier by means of managers keeping track of the state of interaction and protocols to make the various entities communicate with each other.

Powers offered by roles to players have a declarative nature that does not only make them easier to be programmed, but allows the organization to dynamically add and remove powers so to have a restructuring of the roles.

The normative part of our work has to be improved, since, at the moment, only a kind of "implicit" one is present. It can be seen, for example, in the constraints which make possible to play a role only if some requirements are respected. We are also considering possible merge with Jess (in order to use an engine for goals processing), and Jason.

References

1. E. Adam and R. Mandiau. Roles and hierarchy in multi-agent organizations. In *CEEMAS*, pages 539–542, 2005.
2. AOS. JACK Intelligent Agents, The Agent Oriented Software Group (AOS), http://www.agent-software.com/shared/home/, 2006.
3. M. Baldoni, G. Boella, V. Genovese, R. Grenna, and L. van der Torre. How to Program Organizations and Roles in the JADE Framework. In *MATES*, pages 25–36, 2008.
4. M. Baldoni, G. Boella, and L. van der Torre. Modelling the interaction between objects: Roles as affordances. In *Knowledge Science, Engineering and Management, First International Conference, KSEM 2006*, volume 4092 of *LNCS*, pages 42–54. Springer, 2006.
5. M. Baldoni, G. Boella, and L. van der Torre. Interaction between Objects in powerJava. *Journal of Object Technology*, 6(2):7–12, 2007.
6. G. Boella, V. Genovese, R. Grenna, and L. der Torre. Roles in coordination and in agent deliberation: A merger of concepts. *PRIMA 2007*, 2007.
7. G. Boella and L. van der Torre. Organizations as socially constructed agents in the agent oriented paradigm. In *Engineering Societies in the Agents World V, 5th International Workshop (ESAW'04)*, volume 3451 of *LNAI*, pages 1–13, Berlin, 2005. Springer.

8. A. Colman and J. Han. Roles, players and adaptable organizations. *Applied Ontology*, 2007.
9. M. Dastani, B. van Riemsdijk, J. Hulstijn, F. Dignum, and J.-J. Meyer. Enacting and deacting roles in agent programming. In *Procs. of AOSE'04*, pages 189–204, New York, 2004.
10. M. Esteva, D. de la Cruz, and C. Sierra. ISLANDER: an electronic institutions editor. In *AAMAS*, pages 1045–1052, New York, NY, USA, 2002. ACM.
11. M. Fisher, C. Ghidini, and B. Hirsch. Organising computation through dynamic grouping. In *Objects, Agents, and Features*, pages 117–136, 2003.
12. D. Grossi, F. Dignum, M. Dastani, and L. Royakkers. Foundations of organizational structures in multiagent systems. In *Procs. of AAMAS'05*, pages 690–697, 2005.
13. O. Gutknecht and J. Ferber. The madkit agent platform architecture. In *Agents Workshop on Infrastructure for Multi-Agent Systems*, pages 48–55, 2000.
14. M. Hannoun, O. Boissier, J. S. Sichman, and C. Sayettat. Moise: An organizational model for multi-agent systems. In *IBERAMIA-SBIA*, pages 156–165, 2000.
15. J. F. Huebner. J-Moise$^+$ programming organizational agents with Moise$^+$ and Jason. In *http://moise.sourceforge.net/doc/tfg-eumas07-slides.pdf*, 2007.
16. J. F. Huebner, J. S. Sichman, and O. Boissier. S-moise+: A middleware for developing organised multi-agent systems. In O. Boissier, J. A. Padget, V. Dignum, G. Lindemann, E. T. Matson, S. Ossowski, J. S. Sichman, and J. Vzquez-Salceda, editors, *AAMAS Workshops*, volume 3913 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2005.
17. A. Jones and M. Sergot. A formal characterisation of institutionalised power. *Journal of IGPL*, 3:427–443, 1996.
18. C. Madrigal-Mora, E. León-Soto, and K. Fischer. Implementing Organisations in JADE. In *MATES*, pages 135–146, 2008.
19. A. Omicini, A. Ricci, and M. Viroli. An algebraic approach for modelling organisation, roles and contexts in MAS. *Applicable Algebra in Engineering, Communication and Computing*, 16(2-3):151–178, 2005.
20. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: Implementing a bdi-infrastructure for jade agents. *EXP*, 3(3):76–85, 9 2003.
21. M. Schillo and K. Fischer. A taxonomy of autonomy in multiagent organisation. In *Agents and Computational Autonomy*, pages 68–82, 2003.
22. N. Tinnemeier, M. Dastani, and J.-J. C. Meyer. Orwell's nightmare for agents? programming multi-agent organisations. In *Sixth international Workshop on Programming Multi-Agent Systems PROMAS'08*, 2008.
23. E. L. van den Broek, C. M. Jonker, A. Sharpanskykh, J. Treur, and P. Yolum. Formal modeling and analysis of organizations. In *AAMAS Workshops*, pages 18–34, 2005.
24. W. van der Hoek, K. Hindriks, F. de Boer, and J.-J. C. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.
25. F. Zambonelli, N. Jennings, and M. Wooldridge. Developing multiagent systems: The Gaia methodology. *IEEE Transactions of Software Engineering and Methodology*, 12(3):317–370, 2003.

Representing Long-Term and Interest BDI Goals

Lars Braubach and Alexander Pokahr

Distributed Systems and Information Systems,
Computer Science Department, University of Hamburg, Germany
{braubach|pokahr}@informatik.uni-hamburg.de

Abstract. In BDI systems, agents are described using mentalistic notions such as beliefs and goals. According to the intentional stance this helps specifying and understanding complex behavior, because the system is made up of folk psychological concepts that humans naturally tend to use for explaining reasoning and behavior and therefore can easily grasp. To close the gap between the natural usage of the term goal and its operationalization within agent systems, BDI goals should reflect the typical characteristics of goals in the folk psychological sense, which is not completely the case for existing BDI goal representations. Hence, in this paper desirable features of BDI goals are presented and important aspects that are currently not covered in existing specifications are further elaborated. Concretely, the representation and processing of BDI goals is extended supporting also *long-term* and *interest* goals. The usefulness of the newly gained expressivity will be illustrated by an example application, implemented in the Jadex BDI agent system.

1 Introduction

A concise definition of the term goal is extraordinarily hard to find, so that it is used in many psychological and artificial intelligence articles without a strict definition and with partially different meanings [2]. The main difficulty of the definition problem arises from the fact that in order to be useful for a variety of application areas, a definition has to reveal the term's essence without being too strict. Typically, definitions in the context of planning [10] and also w.r.t. specific theories for intentional agents (cf. [7, 20]) tend to be to be too narrow and often reduce the meaning of a goal to a desirable world state that needs to be achieved. A recent attempt in the area of multi-agent systems proposes the following definition: "a goal is a mental attitude representing preferred progressions of a particular multi-agent system that the agent has chosen to put effort into bringing about" [29]. Even though the definition is broader than the ones mentioned before and allows capturing different kinds of goals such as achievement or maintenance, it is already quite restrictive. Firstly, it uses the term "preferred progressions", which is not suitable for all kinds of goals. In case of e.g. avoidance goals [28] doing nothing could be better than doing any of the available actions. Secondly, especially the last part of the definition limits its usability by requiring an agent to put effort into the goal achievement. If an agent has no means to

pursue a goal at some moment in time, that doesn't mean that it cannot possess the goal. It could just sit and wait until it gets the possibility to act towards the goal or just wait passively for its achievement [2].

This paper proposes using a property-based view of goals. The rationale behind this view is to abandon the objective to introduce a strict separation of what is a goal and what is not a goal, but to see goals as a tool for analyzing and specifying systems. Considering a goal by its characteristics may further help understanding how it should be represented and processed and avoids definitions with limited applicability. Note, that a similar procedure led to the agreed upon characterizations of the term agent [23, p. 33], especially the weak/strong notion of agency [31], which is based on the characterizing properties autonomy, reactivity, proactivity, social abilities, and mentalistic notions.

In the next Section 2, the characteristics of BDI goals will be presented. Thereafter, in Section 3 existing goal representations will be extended under consideration of the uncovered aspects of the previous section. In this respect, special attention will be paid to long-term and interest goals, which have in common that they both might not lead to actions immediately respectively at all. In Section 4 the usefulness of the extended goal semantics will be illustrated by a booktrading example application. Finally, in Section 5 a conclusion is given and some aspects of future work are presented.

2 Characteristics of BDI Goals

After having shown the difficulties in defining the term goal precisely, in this section characteristics of goals will be discussed especially in the context of the belief-desire-intention (BDI) model, because it is one of the predominant agent architectures today [25]. In addition to identifying those properties we will examine the degree to which the existing PRS architecture addresses these issues and which deficiencies still exist.

Before concrete characteristics will be presented, the three different BDI perspectives – philosophical, logical and software technical – will be sketched, because they use slightly different terms. In the original work of Bratman [3] the most general meaning of the goal concept in the form of desires is introduced. A desire represents the motivational reasons for an agent's acting. Bratman allows desires to be quite vague and also conflicting so that an agent has to decide to commit to some of its desires making them concrete intentions, which are considered as conflict-free. In contrast to this perspective, in the logical interpretation of [21] no desires but only goals are considered. They are represented as logical formulae of a branching time logic. Here goals are seen as a subset of belief-accessible worlds and are therefore per se declarative and of type achieve. Hence, in this perspective the only difference between goals and beliefs is the optative vs. indicative interpretation of the logical expression. Considering the software engineering perspective the goal concept has been further simplified and reduced to be some kind of volatile event. In the procedural reasoning system (PRS) architecture [9] and AgentSpeak(L) [19] those kinds of goal events are only used for triggering suitable plans and do not posses an explicit representation.

In the following sections an initial attempt is made to identify the most important goal properties from the existing literature. The first five properties have already been identified in a seminal paper of Rao and Georgeff [22]. In addition to these basic properties several further desirable characteristics can be found in the agent as well as social science literature. Note that the further characteristics mainly aim to isolate goal properties that are useful for a software engineering perspective. The discussion in all sections will first explain the meaning of the property and will then discuss its support in the context of the original PRS architecture as well as recent advancements.

Persistent Persistent goals are entities that have a persistent character, which means that they exist over a period of time. In volatile environments it is important for an agent to commit to its goals and give them up only for good reasons. Hence, the persistence of goals serves for stability in an agent's behavior [21].

The persistency of goals intentions has not been defined exactly for the PRS architecture. Instead it has often been discussed in the context of commitment strategies [21, 30, 26], whereby such strategies determine to what extent an agent should keep pursuing its current goals. In the literature a distinction between blind (fanatical), single-minded and open-minded commitment strategies have been proposed. These strategies implement different strengths of commitments. The most committed agent is blindly committed and sticks to its goals until they finally succeed. A single-minded agent also abandons goals on failure and an open-minded agent can get rid off goals by also dropping them intentionally. Experiments have shown that the efficiency of those strategies is heavily dependent on the existing environmental dynamics, i.e. the faster an environment changes the more flexibility an agent has to adapt its goals [13]. Compared to human decision making, especially an open-minded strategy seems to be promising for truly goal-directed agents, because BDI agents should be enabled to reason about their goals (support goal deliberation) and drop them any time, if an important reason occurs.

Consistent In order to describe the consistency property of goals it is necessary to distinguish between the actively pursued goals called adopted goals and the currently inactive goals called candidate goals resp. options [17, 25]. The adopted goals of an agent should be consistent with each other at any point in time in the sense that all goals should be achievable concurrently. An agent should therefore refrain from pursuing a goal, which it thinks stays in conflict with some adopted goals. In case the agent wants to urgently adopt this new goal, a goal deliberation process has to decide if a conflict-free goal set can be found and possibly then has to drop some of the already adopted goals.

The original PRS architecture assumes goals to be always consistent and does not take into account the first phase of practical reasoning, i.e. goal deliberation [16]. This shifts the task for ensuring conflict-freeness to the application layer so that the developer has to cope with these tedious issues directly. This deficiency of the original architecture has been subject of intensive research yielding proposals for supporting also the goal deliberation phase [27, 17].

Possible An adopted goal should be possible to pursue, i.e. an agent should be convinced that it can achieve a goal and it does not contradict its current beliefs.

This property ensures that an agent does not adopt goals it cannot achieve, but it does not guarantee that a goal can always be successfully pursued.

In PRS the conformance of goals and beliefs cannot be directly ensured due to the event-based character of goals. In an indirect way, the pre- and context conditions of plans help guaranteeing that a goal can only be (successfully) processed when those conditions are valid. Nonetheless, as plans could be applicable for different goal types this support is not fully adequate and should be complemented with checks on the goal level.

Known/Explicit A rational agent should be aware of all its goals (candidate and adopted), because this is a necessary prerequisite for any kind of reasoning on its objectives [17].

In PRS an agent knows its goals as long as they are part of the means-end reasoning. The initialization of a goal normally results from a subgoal call within a plan and leads to the generation of a goal event. This event is saved within the corresponding intention of the calling plan, often called the intention stack [19]. During the processing of the goal via different plans the goal is kept in the stack and is e.g. used to save information about its execution state (e.g. which plans have already been tried) [12]. This event-based representation is not expressive enough for supporting goal deliberation and hence explicit goal representations have been proposed [5, 29].

Unachieved An agent should only pursue goals, which it assumes to be unachieved. This kind of behavior will ensure that no unnecessary actions will be initiated and resources will not be wasted.

This property is realized by the PRS architecture via testing the achievement condition of a goal before plan processing is started. In case the condition is immediately true, the goal is considered as succeeded and no plans will be executed. Even though this mechanism ensures correct achieve goal processing, it cannot directly be applied to other goal kinds such as maintain. In order to support this property generically its meaning needs to be adapted to different goal kinds guided by the idea to avoid means-end reasoning if the goal does not require it. E.g. a query goal, that is responsible for information retrieval, should only initiate plan processing when the requested data cannot be directly extracted from the beliefbase [5].

Producible/Terminable In order to be useful for agents, goals should be producible and terminable [8]. For the creation as well as the termination of goals, procedural as well as declarative means should be supported, i.e. an agent should be enabled to create/terminate a goal from a plan as well as due to situational reasons.

In PRS goals can typically be created only in a procedural way by issuing subgoal calls from within a plan. The ex post termination of goals is not possible at all due to their implicit representation. A plan is only allowed to issue one subgoal at a time (intention stack) and the subgoal call itself passivates the original plan and lets it wait until the subgoal processing has finished. Declarative means for creating and terminating goals have been introduced e.g. in [5] and rely on a generic representation for the various goal kinds.

Suspendable In addition to the termination of goals it can be advantageous in certain situations to suspend the pursuit of a goal [8, 5, 24, 29], e.g. if the agent has devoted considerable effort into bringing about the goal and cannot continue to pursue it due to a conflict with another possibly more important goal. Further use cases for goal suspension are detailed in [24]. The suspension of a goal should allow saving the current processing state of that goal and continue from there when it gets activated again [24].

The original PRS architecture does not support the suspension of goals. In [5] and [29] the suspension of goals has been addressed by introducing goal lifecycle states, which allow differentiating between active and suspended goals in an agent. The underlying concept is extended in [24] by also addressing the suspension of plans that are executed whilst their goal is suspended.

Variable Duration Intelligent behavior is based on a combination of strategic and tactical action. Strategic behavior is based on long-term goals, which persist over longer time periods and are typically challenging to achieve, e.g. they need several milestones being reached before the goal as a whole can be tackled. Tactical behavior is in many cases based on short-term goals or even reflexes. Hence, short-term goals often only live for the short moment in which the reason for their creation, e.g. an environmental change, was detected. These kinds of goals are closely linked to (physical) actions and exhibit event-based character.

The PRS architecture focuses exclusively on means-end reasoning and therefore goals only exist during their execution phase, i.e. a goal is held as long as plans are executed for this goal. If no (more) plans are available for a goal the means-end reasoning phase has finished and the goal is considered as finished. This does not necessarily mean that goals are always short-term in PRS, because the corresponding plans could be long-lasting. Nevertheless, using long-term goals requires that goal processing can be immediately started, which is not always desired. Furthermore, goals in PRS are typically not of strategic nature, because conflicts between them cannot be detected and long-lasting goals would considerably increase the probability of goal clashes. Hence, the traditional PRS idea is more centered on realizing short-term goal-driven behavior.

Action Decoupled Goals express and incarnate motivations with respect to a specific situation. This motivation can exist even if an agent cannot contribute actively to the goal achievement. These so called interest or passive goals do not directly lead to action execution, but should nonetheless be allowed to persist within an agent [2]. On the one hand, an agent might eventually gain new procedural knowledge for pursuing the goal [1] or on the other hand the goal might be fulfilled by a third party, e.g. other agents.

Due to the action-centeredness of PRS, interest goals cannot be represented. Instead goals are always part of an intention stack and cannot exist without that structure. The means-end reasoning of PRS ends as soon as no further plans can be executed. This also terminates the corresponding goal by popping the event from the intention stack.

2.1 Challenges

The aforementioned goal properties shape the complex nature of goals and also indicate in which directions the PRS architecture should be further extended

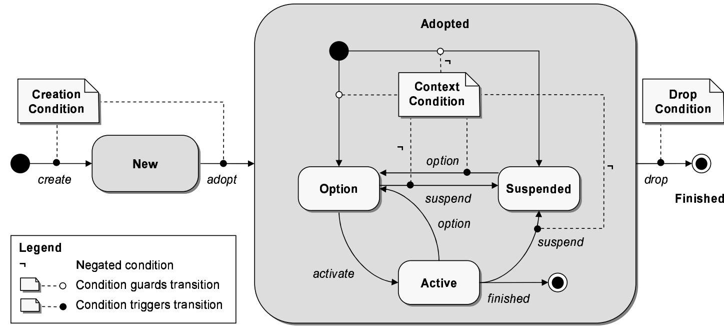


Fig. 1. Basic goal lifecycle

to support the full gamut of goals. Previous works mainly tackled aspects of explicit goal representation [26, 5, 29] and goal deliberation aspects [17, 16, 27]. Goal processing was examined with respect to PRS means-end reasoning [21, 6] and recently also concerning additional features such as goal suspension [24].

To our knowledge none of the existing works has tackled questions of long-term and interest goals for BDI agents, even though these kinds of goals represent a helpful extension for the conceptual canon of BDI agent programmers. The importance of interest goals is emphasized in the literature especially by the cognitive structure of emotions model (OCC - Ortony, Clore and Collins) [14], which assumes that three different goal types exist: "active goals" an agent can directly bring about by performing some actions (e.g. open a bottle), "interest goals" representing states of the world an agent would like to become reality but cannot do anything about pursuing them (e.g. make my soccer team win) and finally "replenishment goals", which repeatedly spawn activities only on demand (e.g. keep healthy). The first and third OCC goal types are already covered by BDI achievement and maintenance goals [5], whereas no support for long-term and interest goals exists. Application cases for interest goals are all scenarios in which external factors (be it actors, processes or sth. else) are responsible for the agent's goal achievement. Examples include conversation handling, where one participant depends on its communication partners [15] and acting in competitive multi-agent settings, where the actions of one agent can contribute to or thwart the goals of another one [11]. Hence, this paper investigates how long-term and interest goals can be represented and how the PRS architecture needs to be modified in order to allow their processing.

3 Long-Term and Interest Goals

To meet the set out requirements and enable a representation of long-term and interest goals in BDI agent systems, two properties should be ensured. First, goals need to be represented explicitly and separately from other BDI constructs such as events or procedural plans. Only with such an explicit representation, the long-term and action-decoupled goals can exist independently of short-lived events and concrete plans resp. actions. Second, an agent should stay committed

to these kinds of goals, even if a goal cannot (immediately) be achieved. Otherwise, the agent will not know when to start acting towards the goal when the time comes (long-term goal) or to refrain from counterproductive actions until the goal is achieved (interest goal).

In the following, an extended goal representation will be introduced that is based on existing work on explicit goal representation and adds the necessary property of *long-term, action-independent commitment.* First, a well-accepted goal lifecycle model from the literature will be described as it forms the basis for the new goal representation. It will be shown, how long-term and interest goals fit into this general model. Second, the detailed processing of long-term and interest goals will be discussed. Moreover, it will be shown how the long-term and interest state of goals can be embedded in different goal types, such as achieve, perform, maintain. The section closes with considerations about the usage of long-term and interest goals.

3.1 Goal Lifecycle Model

Figure 1 shows a basic goal lifecycle introduced in [5]. This lifecycle divides the set of adopted goals of an agent according to the three possible substates *option*, *active* and *suspended* in order to support the goal deliberation and means-end reasoning phases [30]. Thereby, the means-end reasoning of the agent operates on active goals only, i.e. only active goals can lead to the execution of plans and actions. Moreover, suspended goals are goals, which currently cannot be pursued due to an invalid context condition, while options are those goals, which the agent's deliberation mechanism has decided not to pursue (e.g. in favor of other more important goals). Although each goal can only be in exactly one of the substates at any point in time, the state of a goal can change, e.g. when changes happen to beliefs or other goals of an agent.

Based on this generic goal lifecycle, the characteristics of long-term and interest goals can be defined. Whenever a goal enters the active state, the agent will start the means-end reasoning process in order to find suitable means for pursuing the goal. Unlike usual short-term goals, which immediately lead to actions, for a long-term or interest goal it might be an appropriate means to actually do nothing at all. *Therefore, a long-term or interest goal is a goal, which can be active even without executing any plans for it.* The distinction between long-term in contrast to interest goals is merely one of future expectations. For a long-term goal, the agent expects to find suitable means somewhere in the future, while for an interest goal, the agent does not expect to be able to contribute to goal achievement, but still expects that the goal might be (automatically) achieved in the future, if the agent refrains from doing counterproductive actions.

Allowing for goals to be active even without suitable actions leads to some important advantages with respect to goal deliberation processes. Because the goal is among the active goals of an agent, it will be considered as such during the agent's goal deliberation. Therefore, other conflicting goals will not be activated, unless they are considered more important than a currently active long-term or interest goal. This example also shows that sticking to a long-term or interest goal

even without a suitable plan does not contradict an open-minded commitment (cf. section 2), as the agent still can decide to abandon the goal at any time. Another advantage with respect to long-term goals is that for an active goal, the agent can adopt any suitable plan as soon as it becomes available/viable, and does not have to enter complex goal deliberation processes.

3.2 Processing Long-Term and Interest Goals

The previous section showed that the expressibility of the goal lifecycle model can be extended to allow for long-term and interest goals by supporting active goals even when there are (currently) no suitable plans. In general, not all goals are of the long-term/interest type, but usually should be abandoned, when no suitable plans can be found. Therefore, an agent needs to know which goals are of long-term/interest type and how to treat these goals differently from the "normal" short-term goals. The requirements for this special treatment can be condensed to the following questions:

When to stop, but not drop? The usual behavior is to fail a goal, when no suitable plans can be found once the goal has become active. For long-term/interest goals, the agent programmer needs a mechanism for overriding this behavior in such a way that the agent will stop the means-end reasoning, but not drop the goal.

When to continue processing? Some time after a long-term goal has stopped processing, the agent should re-check the availability of plans. For efficiency and effectiveness the programmer should have fine-grained, yet simple control over the re-check activity for ensuring that the agent stays reactive in a changing environment, but avoiding the overhead of unnecessarily checking too frequently.

When to succeed/finally fail? Long-term as well as interest goals should not stay in the agent forever. For one, similar to short-term goals the agent should be able to detect when a goal has been achieved, e.g. because the environment has changed or some successful plan could finally be found. Moreover, because dropping unachievable long-term/interest goals is not done automatically, an agent programmer may want to explicitly state reasons for dropping a long-term or interest goal.

To capture the required functionality, a generic goal processing component is introduced (cf. Figure 2). This component is generic in the sense that it is independent of the concrete goal type (perform, achieve, maintain, etc.), but forms the conceptual and technical basis for processing of all goal types. The component is activated through the *in* edge, which is triggered depending on the goal type, e.g. when a goal becomes active (achieve) or a condition becomes violated (maintain). Regardless of how the component is activated, it will enter two nested loops, which are responsible for basic means-end reasoning and long-term/interest goal handling respectively. In the following sections it will be discussed, how this generic component gives answers to the questions raised above.

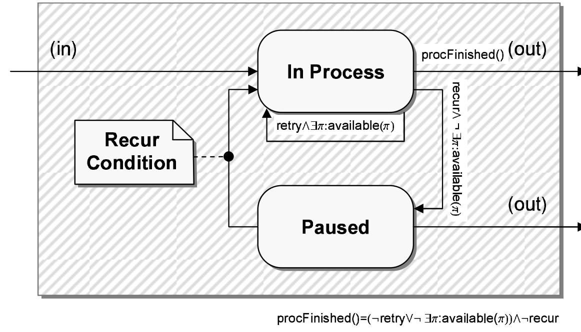


Fig. 2. Generic goal processing component

When to stop, but not drop? The inner loop is the "retry loop", which represents traditional PRS-style means-end reasoning as known from currently implemented BDI systems. This part is captured by the *In Process* state in the figure, in which usually the agent selects and executes a single plan for a goal.[1] If the goal is not finished after the plan has been executed ($procFinished()$), the retry loop continues, leading to the next plan being selected and executed. The retry loop iterates – unless retry behavior is disabled with the *retry* flag – until no more applicable plans are available for the goal ($retry \wedge \exists\pi : available(\pi)$).

To extend this basic means-end reasoning with the required functionality for handling long-term and interest goals, a new *Paused* state is introduced. This state effectively represents the means of "doing nothing" to achieve a goal. Extended means-end reasoning for long-term/interest goals therefore happens according to the outer "recur loop", which alternates between the *In Process* and *Paused* states. The first question posed above then becomes the question of when to move from the *In Process* to the *Paused* state. For this decision, the *recur* flag is introduced that an agent developer can set to true for a goal to be handled as a long-term/interest goal. Hence, the *Paused* state is entered, when the *recur* flag is set and no (more) plans are available ($recur \wedge \neg\exists\pi : available(\pi)$). Note that *Paused* and *In Process* are substates of the lifecycle state *Active* (cf. Figure 1), which means that paused goals suppress the execution of other conflicting, but less important goals e.g. according to the "easy deliberation strategy" [17].

When to continue processing? The continuation of processing forms the second part of the "recur loop", i.e. moving from the *Paused* state back to *In Process*. To allow fine-grained control over when an agent should reconsider the processing of long-term goals, three different specification means are supported – recur delay, recur condition, and recur action – that a developer can choose from or combine, depending on the application at hand. The recur delay of a goal is a simple mechanism for continuously looking for newly applicable plans becoming available. This allows the developer to specify a time interval, after which

[1] For brevity, we do not cover detailed fine-tuning of the means-end reasoning process, such as caching vs. recalculation of the applicable plan list (APL) or parallel execution of plans for the same goal (post-to-all), even though these variations are supported by the proposed model as well.

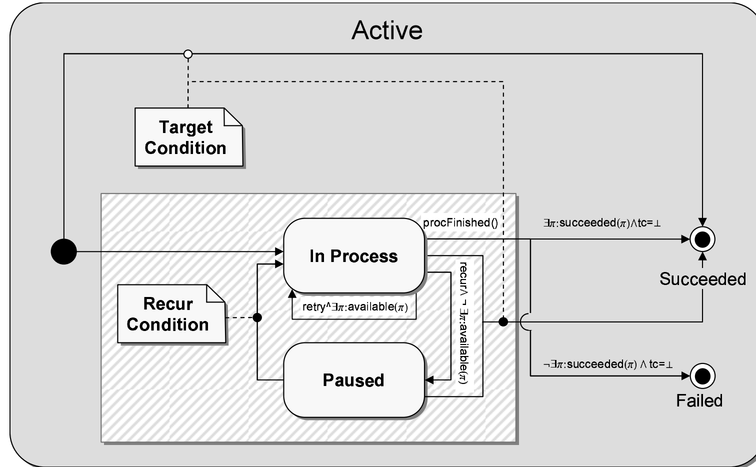


Fig. 3. Extended processing for achieve goals

processing of the goal should be restarted. A more advanced way is the *Recur Condition*, illustrated in the figure. The recur condition allows the specification of a world state (based on the agent's beliefs) that should cause reprocessing of the goal. Therefore it provides a declarative way for specifying the conditions, under which a reconsideration of the goal becomes worthwhile. The most flexible, but least automated way is an explicit recur action that can be manually invoked on a goal. Therefore, the developer can provide arbitrary code (e.g. inside a procedural plan), to determine when certain goals should be reconsidered and explicitly invoke the recur action on these as needed.

When to succeed/finally fail? In Figure 2, the two *out* edges from the *In Process* as well as *Paused* state determine possible ways of exiting the goal processing. First, it should be noted that as illustrated in Figure 2, goal processing would never finish when the *recur* flag is set to true. The lower *out* edge has no guard and therefore will never actively trigger, while the guard on the upper *out* edge is $\textit{procFinished}\,() = (\neg \textit{retry} \vee \neg\exists\pi : \textit{available}\,(\pi)) \wedge \neg \textit{recur}$, i.e. processing is only finished, when *retry* and *recur* are both false or when *recur* is false and no more plans are available. An answer to the question above, therefore cannot be given in the context of the generic goal processing component alone, but also needs to consider the generic goal lifecycle (cf. Figure 1) and the specifics of the different goal types, such as achieve and maintain.

The generic goal lifecycle of Figure 1 allows for several ways of exiting the *Active* state, which would cause the substates like *In Process* and *Paused* to be exited as well. First, the *Active* state can be exited due to goal deliberation issues, e.g. moving to the *Option* state when a more important but conflicting goal occurs or to the *Suspended* state, when the context of the goal becomes invalid. In both cases, processing of the goal would be stopped regardless of the goal being a short-term, long-term, or interest goal. Moreover, goals can be abandoned at any time, e.g. when the *Drop Condition* triggers or the goal is dropped manually. In the latter cases, the goal will finish before being achieved

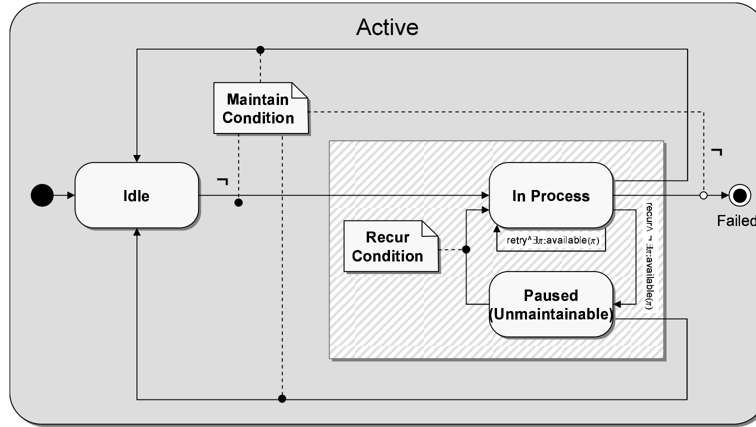


Fig. 4. Extended processing for maintain goals

and therefore can be considered as finally failed. How to determine goal success depends on the specific goal type and will be discussed in the next section.

3.3 Realization of Specific Goal Types

The extended reasoning process has been embedded into the four goal types perform, achieve, query, and maintain, as introduced in [5]. For space reasons, only the achieve and maintain goal types will be discussed in the following. Figure 3 shows the extended model for processing achieve goals. For achieve goals, the generic goal processing component is used as a basis and has been augmented by the *Target Condition*, which can trigger a transition from the *In Process* or *Paused* state to the *Succeeded* state. Moreover, achieve goals without target condition ($tc = \perp$) are also supported by two guards on the upper *out* edge and are considered succeeded, when at least one plan finishes successfully ($\exists \pi : succeeded(\pi) \wedge tc = \perp$) or failed otherwise ($\neg \exists \pi : succeeded(\pi) \wedge tc = \perp$).

For maintain goals (cf. Figure 4), the newly introduced *Paused* state has been mapped to the *Unmaintainable* state of the original processing model from [5]. Unlike the other goal types, a maintain goal does not start with processing immediately. Instead, the goal initially enters the *Idle* state and only moves to *In Process*, when the *Maintain Condition* is violated. While processing is active, a fulfilled maintain condition will move the goal back to the *Idle* state from both *In Process* and *Paused*. This semantics exactly resembles the original semantics from [5] (with *Unmaintainable* renamed to *Paused*), except when the *recur* flag is set to false the maintain goal will stop processing and fail, when none of the available plans is able to re-establish the maintain condition.

3.4 Usage of Long-Term and Interest Goals

The model presented above has been implemented in the Jadex agent framework [18]. One primary advantage of the model is that long-term and interest goals

are represented just like other (short-term) goals. Therefore, their usage does not differ from these and the available mechanisms for creating and handling goals can be reused. In this respect, long-term and interest goals can be given to an agent, when it is born (initial goals) and can also be dynamically created at runtime based on creation conditions or inside plans. Plans can choose to dispatch goals as independent top-level goals that exist outside the scope of the plan or as subgoals, which will be automatically dropped when the plan finishes or is aborted. This control over the goal scope is especially important for long-term and interest goals, which potentially can reside inside an agent for a long period of time. Due to the unified representation of long-term and short-term goals, the plans that get executed in response to a long-term goal do not need to know about the nature of the goal and can be defined independently of the goal's nature. The next section will show how long-term and interest goals can be employed in the context of an illustrative example scenario.

4 Example Application

To illustrate how long-term and interest goals can be used in practice the book-trading scenario from [4] is used, where personal buyer and seller agents are responsible for trading books according to orders given by their principals. The participants use a market-based coordination strategy following the contract-net protocol for reaching agreements acceptable for both sides. It is assumed that buyers take the initiator role of the protocol, whereas sellers play the participant role. An order of a principal includes all relevant data needed for an agent to be able to buy resp. sell a book. Concretely, it contains the name of the book, the start and limit prices as well as a deadline at which the transaction has to be done at latest. The start price represents the acceptable price for an agent at the beginning of the negotiation. While time passes and the deadline approaches a linear price adaptation strategy is used to calculate the currently acceptable price between start and limit price.[2]

Buy or sell orders are entered by the principals through the user interface for each agent. For each of these orders the agents form purchase resp. sell goals, which express the motivations of the principals. For a buyer agent a purchase book goal represents the long-term goal for buying a book, according to the definitions in the order. It is of long-term nature, because initially there might be no seller available that offers the book at the desired price. Nevertheless, the agent should not drop the goal in this case, but instead wait for new sellers to appear or the book gets cheaper at the available sellers. The purchase book goal is therefore modeled as an active achieve goal, which has the purpose of initiating negotiations with potential sellers in fixed time intervals until the book could be bought or the deadline has passed. At the top of Figure 5 the concrete implementation of the *purchase_ book* goal is illustrated. The *purchase_ book* goal (lines 1-5) contains the principal's order in a corresponding parameter (line 2). It

[2] In case of a buyer the start price is lower than the limit price and is continuously increased. The opposite behavior is used by sellers.

```
1  <achievegoal name="purchase_book" recur="true" recurdelay="10000">
2    <parameter name="order" class="Order"/>
3    <targetcondition>Order.DONE.equals($goal.order.state)</targetcondition>
4    <dropcondition>$beliefbase.time > $goal.order.deadline </dropcondition>
5  </achievegoal>
6
7  <achievegoal name="sell_book" recur="true">
8    <parameter name="order" class="Order"/>
9    <targetcondition>Order.DONE.equals($goal.order.state)</targetcondition>
10   <dropcondition>$beliefbase.time > $goal.order.deadline </dropcondition>
11 </achievegoal>
```

Fig. 5. Purchase and sell book goals

is defined as a long-term goal via the recur flag (line 1), which enables the long-term processing loop. In addition, it is made active by specifying the recurdelay, stating that each 10 seconds (10000 ms.) a new negotiation round is started. If the negotiation is successful, the state of the buy order will change to *DONE*, which is tracked by the goal's targetcondition (line 3). On the other hand, the dropcondition (line 4) monitors the deadline and lets the goal automatically fail, when no negotiation result could be achieved before the deadline ends.

The seller's *sell_book* goal (lines 7-11) is realized in a very similar way. The main difference here is that it is a pure interest goal, i.e. it is assumed that sellers passively wait for buy requests to come in and match them with their existing sell goals. An interest goal is specified by activating the recur flag without specifying means for recur initiation, i.e. no recur delay and no recur condition (line 7). The seller agent has no plans for achieving its *sell_book* goal. But the agent does have a plan for reacting to buy book requests from buyer agents and engaging in a corresponding negotiation. When such a negotiation comes to a result, the target condition (line 9) is fulfilled and the *sell_book* goal is achieved.

This example shows how long-term and interest goals can facilitate the high-level and natural modeling of BDI scenarios. The availability of these conceptual abstractions allows for a direct mapping of buy and sell orders to goals, which are present as long as the corresponding orders are relevant. Using only standard BDI goals would require additional error prone code: The buyer's long-term goal could be emulated using a long-term plan, which captures the recur semantics and initiates negotiations in certain intervals. The seller's interest goal would have to be mapped to other structures such as beliefs leading to a rather artificial design, which would also differ a lot from the buyer side.

5 Conclusion

This paper has tackled the representation and processing of long-term and interest goals. In order to understand what make-up goals in BDI agent systems definitions of the term goal have been reviewed. This review mainly revealed that the essence of the term is very hard to capture, because on the one hand different perspectives on BDI exist – ranging from philosophical to implementational – and on the other hand many definitions tend to be too restrictive by overlooking

important goal aspects. As a result, this paper has proposes characterizing goals according to their typical properties, similar to property-based definitions of the term agent. Such a specification is more practically useful as it has the aim of supporting the goal-oriented software specification and is not targeted towards a clear-cut separation of what is a goal and what isn't.

Based on the characterization of goals it has been shown that long-term and interest goals are not currently considered in the modeling and implementation of BDI systems. These goals are typically long-lasting, whereby they can be active without having plans executed for them. In the case of interest goals, an agent does not possess plans for their fulfillment, whereas in the case of long-term goals, plans could exist but not fit to the situation at activation time. Their main relevance can be seen in the more strategic nature of these goals allowing also long-lasting objectives to be expressed. Especially, in combination with goal deliberation mechanisms such strategic goals represent useful extensions to traditional BDI goals, which are of rather tactical, short-term nature.

The main contribution of this paper consists of providing a representation and processing mechanism for long-term and interest goals. The new mechanism builds on the explicit goal representation from [5], and introduces a generic goal processing component for short- and long-term means-end reasoning. This component introduces two control loops, one responsible for traditional plan selection and execution and one responsible for pausing the execution of long-term goals in the case where no processing is currently possible. This generic component can be used to support different goal kinds such as achieve and maintain. The concepts of long-term and interest goals have been implemented within the Jadex BDI agent system and already been used for building different example applications, such as the presented booktrading scenario.

In future work we plan to further extend the expressiveness of goal specifications available for describing BDI agent systems. In this respect one important goal type are soft-goals, which represent non-functional properties and have therefore been excluded from the implementation layer so far.

References

1. Ancona, D., Mascardi, V., Hübner, J., and Bordini, R. Coo-AgentSpeak: Cooperation in AgentSpeak through Plan Exchange. In *Proceedings of AAMAS'04* (2004), ACM press, pp. 698–705.
2. Beaudoin, L. *Goal Processing in Autonomous Agents.* PhD thesis, Mar. 1995.
3. Bratman, M. *Intention, Plans, and Practical Reason.* Harvard Univ. Press, 1987.
4. Braubach, L., and Pokahr, A. Goal-oriented interaction protocols. In *Proceedings of MATES'07* (2007), Springer, pp. 85–97.
5. Braubach, L., Pokahr, A., Moldt, D., and Lamersdorf, W. Goal Representation for BDI Agent Systems. In *Pr. of ProMAS04* (2005), Springer, pp. 44–65.
6. Busetta, P., Howden, N., Rönnquist, R., and Hodgson, A. Structuring BDI Agents in Functional Clusters. In *Proc. of ATAL'99* (2000), Springer, pp. 277–289.
7. Cohen, P. R., and Levesque, H. J. Intention is choice with commitment. *Artificial Intelligence 42* (1990), 213–261.

8. Dignum, F., and Conte, R. Intentional Agents and Goal Formation. In *Proceedings of ATAL'97* (1997), pp. 231–243.
9. Georgeff, M., and Lansky, A. Reactive Reasoning and Planning: An Experiment With a Mobile Robot. In *Proceedings of AAAI'87* (1987), AAAI, pp. 677–682.
10. Ghallab, M., Nau, D., and P.Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers, May 2004.
11. Johns, M., and Silverman, B. G. How Emotions and Personality Effect the Utility of Alternative Decisions: A Terrorist Target Selection Case Study. In *Proceedings of SISO'01* (2001), pp. 55–64.
12. Kinny, D. Algebraic specification of agent computation. *Journal Applicable Algebra in Engineering, Communication and Computing 16*, 2-3 (July 2005), 77–111.
13. Kinny, D., and Georgeff, M. Commitment and effectiveness of situated agents. In *Proceedings of IJCAI'91* (Feb. 1991), pp. 82–88.
14. Ortony, A., Clore, G. L., and Collins, A. *The Cognitive Structure of Emotions*. Cambridge University Press, 1988.
15. Pasquier, P., Dignum, F., Rahwan, I., and Sonenberg, L. Interest-based negotiation as an extension of monotonic bargaining in 3apl. In *PRIMA'06* (2006), Springer, pp. 327–338.
16. Pokahr, A., Braubach, L., and Lamersdorf, W. A Flexible BDI Architecture Supporting Extensibility. In *Proc. of IAT'05* (2005), IEEE, pp. 379–385.
17. Pokahr, A., Braubach, L., and Lamersdorf, W. A goal deliberation strategy for bdi agent systems. In *Proceedings of MATES'05* (2005), Springer, pp. 82–94.
18. Pokahr, A., Braubach, L., and Lamersdorf, W. Jadex: A BDI Reasoning Engine. In *Multi-Agent Programming: Languages, Platforms and Applications* (2005), Springer, pp. 149–174.
19. Rao, A. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In *Proceedings of MAAMAW 1996* (1996), Springer, pp. 42–55.
20. Rao, A., and Georgeff, M. Asymmetry thesis and side-effect problems in linear-time and branching-time intention logics. In *Proc. of IJCAI'91* (1991).
21. Rao, A., and Georgeff, M. BDI Agents: from theory to practice. In *Proceedings of ICMAS 1995* (1995), MIT Press, pp. 312–319.
22. Rao, A. S., and Georgeff, M. P. An abstract architecture for rational agents. In *Proceedings of KR'92* (1992), pp. 439–449.
23. Russell, S., and Norvig, P. *Artifical Intelligence: A Modern Approach*. Prentice-Hall, 2003.
24. Thangarajah, J., Harland, J., Morley, D., and Yorke-Smith, N. Suspending and resuming tasks in bdi agents. In *Proc. of AAMAS'08* (2008).
25. Thangarajah, J., Harland, J., and Yorke-Smith, N. A soft cop model for goal deliberation in a bdi agent. In *Proceedings of CP'07* (sep 2007).
26. Thangarajah, J., Padgham, L., and Harland, J. Representation and Reasoning for Goals in BDI Agents. In *Proc. of ACSC'02*.
27. Thangarajah, J., Padgham, L., and Winikoff, M. Detecting and Avoiding Interference Between Goals in Intelligent Agents. In *Proc. of IJCAI'03* (2003).
28. van Lamsweerde, A. Goal-Oriented Requirements Engineering: A Guided Tour. In *Proceedings of RE'01* (2001), IEEE Press, pp. 249–263.
29. van Riemsdijk, B., Dastani, M., and Winikoff, M. Goals in agent systems: a unifying framework. In *Proceedings of AAMAS'08* (2008), pp. 713–720.
30. Wooldridge, M. *Reasoning about Rational Agents*. MIT Press, 2000.
31. Wooldridge, M. *An Introduction to MultiAgent Systems*. Wiley & Sons, 2001.

Debugging BDI-based Multi-Agent Programs

Mehdi Dastani, Jaap Brandsema, Amco Dubel, John-Jules Ch. Meyer

Utrecht University
The Netherlands
{mehdi,jj}@cs.uu.nl, jaap_b82@hotmail.com, amco@orais.org

Abstract. The development of multi-agent programs requires debugging tools and techniques to find and resolve possible defects in such programs. This paper focuses on BDI-based multi-agent programs, discusses some existing debugging approaches that are developed for specific BDI-based multi-agent programming languages, and proposes a generic and systematic approach for debugging BDI-based multi-agent programs. The proposal consists of an assertion language to specify cognitive and temporal behavior of multi-agent programs and a set of debugging tools. The assertions can be assigned to the debugging tools which will be activated as soon as the execution of a multi-agent program satisfies the assertion.

1 Introduction

Debugging is the art of finding and resolving errors or possible defects, also called bugs, in a computer program. In the context of this paper we divide bugs into three categories: syntax bugs, semantic bugs (logical and concurrent bugs), or design bugs. Design bugs arise before the actual programming and are based on erroneous design of software programs. In contrast to design bugs, both syntax and semantic bugs arise during programming and are related to the actual code of the program. Although syntax bugs are (most of the time) simple typos, which can easily be detected by the program parser (compiler), semantic bugs are, as the name implies, mistakes at the semantic level. Because they often depend on the intention of the developer they can rarely be detected automatically by the program parsers. Therefore, special tools are needed to detect semantic bugs. The ease of the debugging experience is largely dependent on the quality of these debugging tools and the ability of the developer to work with these tools.

A promising approach to develop computer programs for complex and concurrent applications are multi-agent systems. In order to implement multi-agent systems, various agent-oriented programming languages and development tools have been proposed [2]. These agent-oriented programming languages facilitate the implementation of individual agents and their interactions. A special class of these programming languages aims at programming BDI-based multi-agent systems, i.e., multi-agent systems in which individual agents are programmed in terms of cognitive concepts such as beliefs, events, goals, plans, and reasoning rules [13, 8, 3, 7].

Despite numerous proposals for BDI-based multi-agent programming languages, there has been little attention on building effective debugging tools for *BDI-based* agent-oriented programs. The existing debugging tools for BDI-based programs enable the observation of program traces (the sequence of program states generated by the program's execution) [5, 8, 6, 7, 3] and browsing through these traces, allowing to run multi-agent programs in different execution modes by for example using breakpoints and assertions [5, 7, 3, 8], observing the message exchange between agents and checking the conformance of agents' interactions with a specific communication protocol [4, 12, 8, 5, 9, 10]. Although most proposals are claimed to be applicable to other BDI-based multi-agent programming languages, they are presented for a specific multi-agent platform and the corresponding multi-agent programming language. In these proposals, debugging multi-agent aspects of such programs are mainly concerned with the interaction between individual agents and the exchanged messages. Finally, the temporal aspects of multi-agent program traces are only considered in a limited way and not fully exploited for debugging purposes.

In this paper, we focus on semantic bugs in BDI-based multi-agent programs and propose a generic approach for debugging such programs. Our proposal extends previous ones by debugging not only the interaction between individual agents in terms of exchanged messages, but also debugging relations between internal states of different individual agent programs.[1] Moreover, we propose a set of debugging constructs that allow a developer to debug both cognitive and *temporal* aspects of the multi-agent program traces. For example, the debugging constructs allow a developer to log specific parts of the cognitive state of individual agent programs (e.g., log the beliefs, events, goals, or plans) from the moment that specific condition holds, stop the execution of multi-agent programs whenever a specific cognitive condition holds, or check whether a trace of a multi-agent program (a sequence of cognitive states) satisfies a specific (temporal) property. In general, we propose a set of debugging actions/tools and an assertion language. The expressions of the assertion language are assigned to the proposed actions/tools such that they are performed/activated when their associated assertions hold during the execution of multi-agent programs. Our approach does not assume a specific representation for the internals of individual agents and can be applied to any BDI-based multi-agent programming language. We only assume that the state of individual BDI-based agents consists of cognitive components such as beliefs, goals, plans, and events/messages without assuming how these components are represented.

In section 2, we discuss some related works on debugging multi-agent programs, and in section 3, we present our generic vision on multi-agent programs and their semantics. Based on this vision, our approach for debugging multi-agent programs is presented in section 4. The paper concludes with some comments and future works.

[1] A developer/debugger of a multi-agent program is assumed to have access to the multi-agent program code and therefore to the internal state of those programs.

2 Background and Related Work

A well-known technique often used for debugging single sequential and concurrent programs is a *breakpoint*. A breakpoint is a marker that can be placed in the program's code. Breakpoints can be used to control the program's execution. When the marker is reached program execution is halted. Breakpoints can be either conditional or unconditional. Unconditional breakpoints halt the program execution when the breakpoint marker is reached. Conditional breakpoints only halt the program execution when the marker is reached and some extra condition is fulfilled. Another (similar) functionality, that can be used to re-synchronize program executions, is called a process barrier breakpoint. Process barrier breakpoints are much like normal breakpoints. The difference is they halt the processes that reached the barrier point until the last process reaches the barrier point. A different debugging technique used for traditional programming practices is called the *watch*. The watch is a window used to monitor variables' values. Most watch windows also allow the developer to type in a variable name and if the variable exists the watch will show the variable's value. In the IDEs of most high-level programming languages the watch is only available when the program's execution is halted. Other traditional debugging techniques are logging and visualization. Logging allows a developer to write some particular variable's value or some statement to a logging window or a file. Visualization is particularly helpful in the analysis and fine tuning of concurrent systems. Most relevant in light of our research is the ability to visualize the message queue.

These traditional debugging techniques have inspired many agent researchers to develop debugging frameworks for multi-agent programs. An example of such a framework, proposed by Collier [5], is designed for Agent Factory and its corresponding AFAPL (Agent Factory Agent Programming Language). In this approach, debugging can be done both at compile time and at run time. At compile time, syntax errors as well as reference to non-existing files, the definition of belief terms, and the declaration of actions and plans are checked. Run time debugging focuses on semantic bugs such as sending wrong messages and the internal working of individual agents. The run time debugging is done by a debugging tool called *AFAPL Debugger*. This tool provides a number of views of an agent system, among which, a view that enables to inspect and trace an agent's internal state, and to start, stop, and step through the execution of the agent program. Beside these standard functionalities, this view enables to highlight the interplay between different mental attitudes of one individual agent, to check the performance of an agent with respect to the execution of perceptors and actuators, and to increase the level of control over granularity of the step operation using breakpoints. Using this extended view tool, one can inspect beliefs that are generated by a perceptor, the current primary commitments of the agent, and a list of all messages sent/received by the agent. The views can be filtered to, for example, focus on the beliefs generated by a single preceptor. Other views present information about services that are deployed on the agent platform and the previous runs (histories) of the agent system.

Sudeikat [11] presents an assertion based debugging mechanism for the Jadex platform and its corresponding programming language. In this approach, assertions statements can be annotated to the BDI elements (e.g., beliefs, goals, plans) of an individual agent program. Assertions specify relations between BDI concepts or the invariant properties of the execution of an agent program. Assertion statements, which evaluate to boolean values, are arbitrary Java statement executed by the underlying Jadex assertion mechanism. When an assertion is evaluated to false, a warning is generated to inform the developer about the agent and the element where the assertion evaluated to false. The execution of the Java statements is triggered by the Jadex BDI reasoning events. They also propose a three dimensional graph of the overall communication structure of the multi-agent system. Finally, they propose run time monitoring of exchanged messages in order to detect possible violation with respect to a given communication protocol.

An important aspect of debugging multi-agent programs is related to message exchanges between individual agents. In many existing approaches, e.g., [1, 7, 3, 8], message exchanges are logged and presented by means of different visualization techniques. In [4], Botía and his colleagues use traditional data mining techniques to visualize the logged (FIPA) exchanged messages. This approach creates two main types of graphs: an agent communication graph, and a clustered graph where agents are clustered based on similar communication activity or cooperation activity. In another work [12], Vigueras and Botía propose the use of causality graphs to track causality amongst messages sent by individual agents. The causality graph is created from conversations that are previously logged. The ordering of the messages is done by using a logical vector clock.

Yet another approach to debug a multi-agent program based on comparing the actual behavior of the program with the desired behavior is proposed by Lam [6]. In this paper, a tracing method and tracer tool are proposed. The tracing method captures dynamic runtime data by logging actual agent behavior. The data is logged by introducing logging statements into the agent program. The captured data is used to create behavioral models of the agents' activities in terms of agent concepts (e.g. beliefs, goals, and intentions). These models can be used to compare the actual behaviour of the models with the expected agent behaviour, to identify bugs in the agent's program. Currently the comparision has to be made manually, since no specification for expected agent behavior has been developed yet. The tracer tool creates relational graphs which can be used to manually verify the initial design diagrams.

The techniques mentioned above are helpful when errors manifest themselves directly to the developer or user. However, errors in a program do not always manifest themselves directly. For mission and industrial critical systems it is necessary to extensively test the program before deploying it. This testing should remove as many bugs (and possible defects) as possible. However, it is infeasible to test every single situation the program could be in. A testing approach proposed for multi-agent programs is proposed by Poutakidis and his colleagues [9, 10].

3 Programming Multi-Agent Systems

A multi-agent program is the declaration of a set of agents, possibly together with the specification of organizational structures and laws that should be respected by the declared agents during their executions. In order to keep our approach generic, we do not make any assumption about organizational structures and laws. So, without losing generality, we assume that a multi-agent program looks like the following program.[2]

```
Agents:         cleaner   :  cleaner.prog    1
                explorer  :  explorer.prog   3
Environment:    gridworld
```

This program declares one cleaner agent and three explorer agents that should cooperate to clean bombs from a grid-like environment, called `gridworld`. This multi-agent program indicates that agents can perform actions in the `gridworld` environment. For this program, we assume that the goal of the explorer agent is to explore the environment and find the bombs, which are placed in that environment. When a bomb is found, the explorer agent communicates the location of the bomb to the cleaner agent who has to dismantle the bomb.

An individual BDI-based agent can be programmed by specifying its initial (cognitive) state/configuration in terms of beliefs (information), events (observation), goals (objectives), plans (means), and reasoning rules (for generating plans). In programming terminology, these ingredients can be considered as (cognitive) data structures specifying the (initial) state/configuration of the agent program. Without losing generality and committing to a specific knowledge representation scheme, we assume in the rest of the paper a BDI-based agent programming language that provides (cognitive) data structures to represent the initial cognitive state/configuration of each individual agent.

The execution of a BDI-based multi-agent program is the concurrent executions of all individual agent programs. The execution of each individual agent program is based on a cyclic process called *deliberation cycle* (sense-reason-act cycle). Each iteration of this process starts with sensing the environment (i.e., receive events and messages), reasoning based on its state (i.e., update beliefs and goals based on events and messages, and generate plans to either achieve goals or to react to events), and performing actions (i.e., perform actions of the generated plans). An execution of a multi-agent program generates a trace (a sequence of multi-agent program states). Each state in such a trace consists of the states of all individual cognitive agents, i.e., it consists of beliefs, events, goals, plans of all individual agents. It is important to note that similar BDI ingredients and deliberation cycle are used in existing BDI-based programming languages such as Jason [3], 2APL [7], Jadex [8], and Jack [13].

[2] It should be noted that the declaration of agents can also be done by means of loading agents in a multi-agent platform such that there is no need for a specific multi-agent program.

In the following, we assume that an execution of a multi-agent program starts with the initial state of the declared agents (specified by the individual agent programs) and generates a sequence of states based on a deliberation cycle (i.e., sense, reason and act cycle). Formally, a state of a multi-agent program is a tuple $\langle A_1, \ldots, A_n, \chi \rangle$, where $A_i = \langle i, \sigma, \gamma, \Pi, \xi \rangle$ is the state of individual agent i (with beliefs σ, goals γ, plans Π, and events ξ) and χ is the environment in which agents' actions can be performed. An execution of a multi-agent program is then a sequence $s_0, s_1, \ldots$, where $s_0 = \langle A_1^0, \ldots, A_n^0, \chi^0 \rangle$ is the initial state specified by a multi-agent program, and state s_n is reached by means of a deliberation action (update beliefs, generate/execute plans, process events) performed by one of the agents in state s_{n-1}.

For example, the multi-agent program mentioned above specifies the initial state $\langle cleaner, explorer_1, explorer_2, explorer_3, gridworld \rangle$. The state of the cleaner agent is $cleaner = \langle c, \sigma_c, \gamma_c, \Pi_c, \xi_c \rangle$, where σ_c represents the beliefs of the cleaner agent (initially specified by `cleaner.prog`), γ_c represents its goal base, Π_c represents its plan base, and ξ_c represents its event base. The beliefs can be a set of propositions, a set of objects, or any other data structure that can represent facts. The exact nature of beliefs depends on the programming constructs provided by the programming language. Note that Π is a set of plans each consists of domain actions, e.g., $\Pi = \{goto(2,3); pickup()\ ,\ sense(bombs)\}$ consists of two plans; the first plan indicates that the agent should move to location $(2,3)$ followed by a picking up (a bomb) action, and the second plan indicates that the agent should sense the world to find some bombs. The state of other agents are similar. The *gridworld* is assumed to be an specification of the state of the `gridworld` environment.

4 Debugging Multi-Agent Programs

Given an execution of a multi-agent program, one may want to check if an agent drops a specific goal when it is achieved, when two or more agents have the same beliefs, whether the number of agents is suited for the environment (e.g. it is useless to have a dozen explorers on a small area, or many explorers when there is only one cleaner that cannot keep up with them.), whether the protocol is suited for the given task (e.g. there might be a lot of overhead because facts are not shared, and therefore, needlessly rediscovered), whether important beliefs are shared and adopted, or rejected, once they are received. We may also want to check if unreliable sources of information are ignored, whether the actions of one agent are rational to take based on the knowledge of other agents, or if sent messages are received by the recipient. This can, for example, be used to locate deadlocks where one more agents keep waiting for a message to be sent.

Ideally one would specify a property by creating an assertion and get notified when the assertion evaluates to true. Similar ideas are proposed in Jadex [8]. In the following, we introduce an assertion language, called MDL (multi-agent description language), to specify the cognitive and temporal behavior of BDI-based multi-agent programs. An MDL assertion is evaluated against the (finite)

trace of a multi-agent program and can activate some debugging tools when they are evaluated to true. The debugging tools are inspired by traditional debugging tools, extended with the functionality to verify a multi-agent program trace. One example of such a debugging tool is a multi-agent version of the breakpoint. The breakpoint can halt a single agent, a group of agents or the complete multi-agent program. This multi-agent version of the breakpoint can also have a MDL assertion as a condition, making it a conditional breakpoint.

4.1 Syntax of Assertion language

In this section, we present the syntax of the MDL written in EBNF notation. An expression of this language describes a property of a multi-agent program execution and can be used as assertions based on which debugging actions/tools will be performed/activated. In the following, ⟨*group_id*⟩ is a group identifier (uncapitalized string), ⟨*agent_id*⟩ an agent identifier (uncapitalized string), ⟨*query_name*⟩ a property description name (a reference to an assertion used in the definition of macros; see later on for a discussion on macros), ⟨*Var*⟩ a variable (Variables are capitalized strings), [*all*] indicates the group of all agents, and ⟨*agent_var*⟩ an agent identifier, a group identifier, or a variable. In order not to make any assumption about the exact representation of an agent's beliefs, goals, events, and plans, we assume *Bquery*, *Gquery*, *Equery*, and *Pquery* to denote an agent's Beliefs, Goals, Events, and Plans, respectively. This makes it possible to apply this assertion language to other BDI-based multi-agent programming languages.

```
<group_def>    ::= "[" <group_id> "]" " = " <agent_list>
<agent_list>   ::= "[" <agent_id> ("," <agent_id>) * "]"
<mdl_pd>       ::= <query_name> "{" <mdl_query> "}"
<mdl_query>    ::= "{" <mdl_query> "}"
                 | <agent_var> "@Beliefs(" <Bquery> ")"
                 | <agent_var> "@Goals(" <Gquery> ")"
                 | <agent_var> "@Plans(" <Pquery> ")"
                 | <agent_var> "@Events(" <Equery> ")"
                 | <UnOp> <mdl_query>
                 | <mdl_query> <BinOp> <mdl_query>
                 | "?" <query_name>
<BinOp>        ::= "and" | "or" | "implies" | "until"
<UnOp>         ::= "not" | "next" | "eventually" | "always"
<agent_var>    ::= <Var> | <agent_id> | <group_id> | "[all]"
```

Note that ⟨*mdl_pd*⟩ is an assertion that describes the (temporal and cognitive) behavior of a multi-agent program execution. In order to illustrate the use of this assertion language, we present a number of examples in which logic-based representation are used to express an agent's beliefs, goals, events, and plans. For example, `bomb(2,3)` (read as *there is a bomb at position (2,3)*) and `clean(gridworld) and carry(bomb)` (read as *the gridworld is clean and*

the agent carries a bomb) are used to represent an agent's beliefs or goals, `event(bombAt(3,4))` (read as *it is perceived that a bomb is at position (3,4)*) or `message(explorer, inform, bombAt(2,3))` (read as *a message is received from explorer informing there is a bomb at position (2,3)*) are used to represent an agent's events and messages, and `goto(X, Y); dropBomb(X',Y')` (read as *go first to position (X,Y) and then drop the bomb that is originally found at position (X',Y')*) are used to represent an agent's plan.

In order to specify that either `all` agents believe that there is a bomb at position `2,3` (i.e., `bomb(2,3)`) or `all` agents believe that there is no bomb at that position (i.e. `not bomb(2,3)`), we can use the following assertion.

```
[all]@Beliefs( bomb(2,3) ) or [all]@Beliefs( not bomb(2,3) )
```

We can generalize this assertion by assigning a name to it and parameterizing the specific beliefs (in this case `bomb(X,Y)`). This generalization allows us to define an assertion as a macro that can be used to define more complex assertions. For example, consider the following generalization (macro) that holds in a state of a multi-agent program if and only if either all agents believe the given belief ϕ or all agents do not believe ϕ.

```
isSharedBelief(φ) { [all]@Beliefs( φ ) or [all]@Beliefs( not φ ) }
```

Note that `isSharedBelief(φ)` can now be used (e.g., in other assertions) to check whether or not ϕ is a shared belief. In general, one can use the following abstract scheme to name an MDL assertion. Parameters `Var1, Var2,` and `Var3` are assumed to be used in the MDL assertion.

```
name( Var1, Var2, Var3, ...) { MDL assertion }
```

The following example demonstrates the use of macros. To use a MDL assertion inside another one, the macro's names should be preceded by a "?" mark. We now define a cell as detected when agents agree on the content of that cell. We define `detectedArea(R)` as follows.

```
detectedArea(X, Y) { ?isSharedBelief( bomb(X,Y) ) }
```

The next example shows a MDL assertion that can be used to verify whether the `gridworld` will eventually be clean if an agent has the goal to clean it. In particular, the assertion states that if an agent `A` has the goal to clean the `gridworld` then eventually that agent `A` will believe that the `gridworld` is clean.

```
 cleanEnvironment(A) {
   A@Goals(clean(gridworld)) implies eventually A@Beliefs(clean(gridworld))
}
```

The following MDL assertion states that an agent `A` will not unintentionally drop the bomb that it carries. More specifically, the assertion states that if an agent believes to carry a bomb, then the agent will believe to carry the bomb until it has a plan to drop the bomb. It is implicitly assumed that all plans will be successfully executed.

```
doesNotLoseBomb(A) {
  always ( A@Beliefs(carry(bomb))
           implies
           ( A@Beliefs(carry(bomb)) until A@Plans(dropBomb(X,Y)) ) )
}
```

4.2 Semantics

The semantics of the MDL language describe how an assertion is evaluated against a trace of a BDI-based multi-agent program. In the context of debugging, we consider *finite traces* generated by partial execution of multi-agent programs (a partial execution of a program starts in the initial state of the program and stops after a finite number of deliberation steps). A finite trace is a finite sequence of multi-agent program states in which the state of each agent is a tuple consisting of beliefs, goals, events, and plans. In the following, we write σ_i to denote the belief base σ of individual agent i. Similar notation will be used for goal base, plan base, and event base.

Definition 1. *Let $s = \langle A_1, \ldots, A_n, \chi\rangle$ be a state (configuration) of a multi-agent program, and $A_i = \langle i, \sigma_i, \gamma_i, \Pi_i, \xi_i\rangle$ be the state of the individual agent i. The assignment functions V_b, V_g, V_e, and V_p determine the beliefs, goals, events, and plans of an individual agent in a state of a multi-agent program. These assignment function are defined as follows: $V_b(i, s) = \sigma_i$, $V_g(i, s) = \gamma_i$, $V_p(i, s) = \Pi_i$, and $V_e(i, s) = \xi_i$.*

An arbitrary MDL assertion can be evaluated with respect to a finite multi-agent program trace that is resulted by a partial execution of a multi-agent program. In the following, we use t to denote a finite trace, $|t|$ to indicate the length of the trace t (a natural number; a trace consists of 1 or more states), st to indicate a trace starting with state s followed by the trace t, $|st| = 1 + |t|$, and functions *head* and *tail*, defined as follows: $head(st) = s$, $head(t) = t$ if $|t| = 1$, $tail(st) = t$ and $tail(t)$ is undefined if $|t| \leq 1$ (*tail* is a partial function). Moreover, given a finite trace $t = s_1 s_2 \ldots s_n$, we write t_i to indicate the suffix trace $s_i \ldots s_n$.

Definition 2. *Let $t = s_1 s_2 \ldots s_n$ be a finite trace of a multi-agent program such that $|t| \geq 1$. The satisfaction of MDL expressions by the trace t is defined as follows:*

$t \models \texttt{i@Beliefs}(\phi) \Leftrightarrow \phi \in V_b(i, head(t))$
$t \models \texttt{i@Goals}(\phi) \Leftrightarrow \phi \in V_g(i, head(t))$
$t \models \texttt{i@Plans}(\phi) \Leftrightarrow \phi \in V_p(i, head(t))$
$t \models \texttt{i@Events}(\phi) \Leftrightarrow \phi \in V_e(i, head(t))$
$t \models \phi \texttt{ and } \psi \Leftrightarrow t \models \phi \text{ and } t \models \psi$
$t \models \phi \texttt{ or } \psi \Leftrightarrow t \models \phi \text{ or } t \models \psi$
$t \models \phi \texttt{ implies } \psi \Leftrightarrow t \models \phi \text{ implies } t \models \psi$
$t \models \texttt{not } \phi \Leftrightarrow t \not\models \phi$
$t \models \texttt{next } \phi \Leftrightarrow tail(t) \models \phi \text{ and } |t| > 1$
$t \models \texttt{eventually } \phi \Leftrightarrow \exists i \leq |t| \quad (t_i \models \phi)$
$t \models \texttt{always } \phi \Leftrightarrow \forall i \leq |t| \quad (t_i \models \phi)$
$t \models \phi \texttt{ until } \psi \Leftrightarrow \exists i \leq |t| \quad (t_i \models \psi \text{ and } \forall j < i \quad (t_j \models \phi))$

Based on this definition of MDL assertions, we have implemented some debugging tools that are activated and updated when their corresponding MDL assertion holds

in a partial execution of a multi-agent program.[3] These debugging tools are described in the next section.

4.3 Multi-Agent Debugging Tools

This section presents the Multi-Agent Debugging Tools (MADTs). In order to use the debugging tools, markers are placed in the multi-agent programs to denote under which conditions which debugging tool should be activated. A marker can consists of a (optional) MDL assertion and a debugging tool. The MDL assertion of a marker specifies the condition under which the debugging tool of the marker should be activated. In particular, if the MDL assertion of a marker evaluates to true for a given finite trace/partial execution of a multi-agent program, then the debugging tool of the marker will be activated. Besides a MDL assertion, a marker can also have a group parameter. This group parameter specifies which agents the debugging tool operates on. The general syntax of a marker is defined as follows:

$\langle marker \rangle$ ::= "`MADT(`" $\langle madt \rangle$ [",” $\langle mdl_query \rangle$][",@" $\langle group \rangle$] ")"

$\langle group \rangle$ ::= "[" $\langle group_id \rangle$ "]" | $\langle agent_list \rangle$

The markers that are included in a multi-agent program are assumed to be processed by the interpreter of the corresponding multi-agent programming language. In particular, the execution of a multi-agent program by the interpreter will generate consecutive states of a multi-agent program and, thereby, generating a trace. At each step of the trace generation (i.e., at each step where a new state is generated) the interpreter evaluates the assertion of the specified markers with respect to the finite trace (according to the definition of the satisfaction relation; see definition 2) and activates the corresponding debugging tools if the assertions are evaluated to true. This means that the trace of a multi-agent program is verified after every change in the trace. This mode of processing markers is called *continuous mode* as it does not stop the execution of the multi-agent program; markers are processed during the execution of the program. An alternative mode of processing markers would be *post mortem*. In this alternative mode, a multi-agent program can be executed and stopped after some deliberation steps. The markers can then be processed based on the finite trace generated by the partial execution of the program. The following example illustrates the use of a marker in a multi-agent program:

```
Agents:        cleaner   :  cleaner.prog    1
               explorer  :  explorer.prog   3
Environment:   gridworld

Markers:       MADT(breakpoint_madt, eventually cleaner@Beliefs(bomb(X,Y)) )
```

In this marker, a breakpoint will be activated as soon as the cleaner agent believes that there is a bomb in a cell. Examples of debugging tools are breakpoint, logging, state overview, or message list. These debugging tools are explained in the rest of this section. It is important to note that if no MDL assertion is given in a specified marker, then the debugging tool of the marker will be activated after each update of the trace. Moreover, if no group parameter is given in the marker, the "[all]" group is used by default.

[3] It should be noted that this definition of the satisfaction relation can behave different than the standards definition of satisfaction relation of LTL which is defined on infinite traces.

Breakpoint The breakpoints for multi-agent programs are similar to breakpoints used in concurrent programs. They can be used to pause the execution of a single agent program, a specific group of agent programs, or the execution of the entire multi-agent program. The example below demonstrates the use of a conditional breakpoint on the agents `explorer1` and `explorer2`. The developer wants to pause both agents as soon as agent `cleaner` has the plan to go to cell `(5, 5)`.

```
MADT(breakpoint_madt,
     eventually cleaner@Plans(goto(5, 5)), @[explorer1, explorer2])
```

Note that it is possible to use the cognitive state of more than one agent as the break condition. The next example demonstrates how a developer can get an indication about whether the number of explorer and cleaner agents are suitable for a certain scenario. In fact, if there are not enough cleaners to remove bombs, or when all explorers are located at the same area, then all explorers will find the same bomb.

```
MADT(breakpoint_madt, eventually [explorers]@Beliefs(bomb(X,Y)))
```

The breakpoint tool is set to pause the execution of all agents, once all agents that are part of the "explorers" group have the belief that a bomb is located at the same cell `(X,Y)`. Note that it need not be explicitly defined to pause the execution of *all* agents. The breakpoint is useful in conjunction with the watch tool to investigate the mental state of the agent. Other agent debugging approaches, e.g., [5], propose a similar concept for breakpoints, but for a single BDI-based agent program. Also, Jason [3] allows annotations in plan labels to associate extra information to a plan. One standard plan annotation is called a breakpoint. If the debug mode is used and the agent executes a plan that has a breakpoint annotation, execution pauses and the control is given to the developer, who can then use the step and run buttons to carry on the execution. Note that in contrast with other approaches, the condition in our approach may contain logic and temporal aspects.

Watch The watch can display the current mental state of one or more agents. Furthermore, the watch allows the developer to query any of the agents' bases. The developer can, for example, use the watch to check if a belief follows from the belief base. It is also possible to use a MDL assertion in the watch; if the assertion evaluates to *true*, the watch will show the substitution found. The watch tool can also be used to visualize which agents have shared or conflicting beliefs. The watch tool is regularly used in conjunction with a conditional breakpoint. Once the breakpoint is hit, the watch tool can be used to observe the mental state of one or more agents. In general, the watch tool should be updated unconditionally and for all agents in the system. Adding `MADT(watch_madt)` to a multi-agent program will activate the watch on every update of its execution trace. In Jason [3], a similar tool is introduced which is called the mind inspector. This mind inspector, however, can only be used to observe the mental state of individual agents. Jadex [8] offers a similar tool called the BDI-inspector which allows visualization and modification of internal BDI-concepts of individual agents.

Logging Logging is done by the usage of probes which, unlike breakpoints, do not halt the multi-agent program execution. When a probe is activated it writes the current state of a multi-agent program, or a part of it, to a log screen or a file (depending on the type of probe). Using a probe without a MDL assertion and without a group

specification is very common and can be done by adding `MADT(probe_madt)` in multi-agent programs. The probe will be activated on every update of the program trace such that it keeps a log of all multi-agent program states. The next example saves the state of the multi-agent program when the cleaner agent drops a bomb in a depot, but there is still some agent who believes the bomb is still at its original place.

```
MADT(probe_madt,
     eventually(cleaner@Plans(dropBomb(X,Y)) and A@Beliefs(bomb(X,Y)))
)
```

Similar work is done in Jadex [11] where a logging agent is introduced to allow collection and viewing of logged messages from Jadex agents. It should be noted that the probes in our approach offer the added functionality of filtering on a cognitive condition of *one ore more agents*.

Message-list Another visualization tool is the message-list, which is one of the simplest forms of visualization. The message-list keeps track of the messages sent between agents, by placing them in a list. This list can be sorted on each of the elements of the messages. For example, sorting the messages on the "sender" element can help finding a specific message send by a known agent. Besides ordering, the list can also be filtered. For example, we could filter on "Senders" and only show the message from the sender with the name "cleaner". To update the message-list on every update of the trace, we can place the marker `MADT(message_list_madt)` in the multi-agent program. Another use of the message-list could be to show only the messages from within a certain group, e.g., `MADT(message_list_madt, @[explorers])` can be used to view the messages exchanged between the members of the explorers group. Finally, in our proposal one can also filter exchanged messages based on conditions on the mental states of individual agents. For example, in the context of our gridworld example, one can filter useless messages, i.e., messages whose content are known facts. Note that exchanging too many useless messages is a sign of non-effective communication. The example below triggers the message list when an agent `A`, who believes there is a bomb at coordinates `X,Y`, receives a message about this fact from another agent `S`.

```
MADT(message_list_madt,
     eventually( A@Beliefs(bomb(X,Y)) and A@Messages(S,P,bombAt(X,Y)) )
)
```

All existing agent programming platforms offer a similar tool to visualize exchanged messages. The main difference with our approach is the ability to log when certain cognitive conditions hold.

Causal tree The causal tree tool shows each message and how it relates to other messages in a tree form. The hierarchy of the tree is based on the relation between messages (replies become branches of the message they reply to). Messages on the same hierarchical level, of the same branch, are ordered chronologically. The advantage of the causal tree (over the message-list) is that it is easier to spot communication errors. When, for example, a reply is placed out of context (not in relation with its cause) this implies there are communication errors. The causal tree also provides an easy overview to see if replies are sent when required. The causal tree tool can be used by adding the marker `MADT(causal_tree_madt)` to multi-agent programs. Another example could

be to set the group parameter and only display message from a certain group, e.g., `MADT(causal_tree_madt, @[explorers])`. It should be noted that for the causal tree to work, the messages need to use performatives such as inform and reply.

Sequence diagram The sequence diagram is a commonly used diagram in the Unified Modeling Language (UML) or its corresponding agent version (AUML). An instantiation of a sequence diagram can be used to give a clear overview of (a specific part of) the communication in a multi-agent program. They can help to find irregularities in the communication between agents. The sequence diagram tool can be used in our approach by adding the marker `MADT(sequence_diagram_madt)` to multi-agent programs. This example updates the sequence diagram on every update of the trace. Another example could be to use the group parameter and only update the sequence diagram for the agents in a certain group, e.g., `MADT(sequence_diagram_madt, @[cleaner, explorer2])`. Adding this marker to our multi-agent program will show the communication between the agents "cleaner" and "explorer2". The sequence diagram tool is useful in conjunction with a conditional breakpoint and the stepwise execution mode where the diagram can be constructed step by step. The sequence diagram is also useful in conjunction with the probe. The probe can be used to display detailed information about the messages. Similar tools are proposed in some other approaches, e.g., the sniffer agent in [1]. However, we believe that the sequence diagram tool in our approach is more effective since it can be used for specific parts of agent communication.

Visualization Sometimes the fact that a message is sent is more important than the actual contents of the message. This is, for example, the case when a strict hierarchy forbids certain agents to communicate. In other cases it can be important to know how much communication takes place between agents. For such situations the *dynamic agent communication* tool is a valuable add-on. This tool shows all the agents and represents the communication between the agents by lines. When agents have more communication overhead the line width increase in size and the agents are clustered closer together. This visualization tool, which can be triggered by adding the marker `MADT(dynamic_agent_madt)` to multi-agent program, is shown in figure 1.

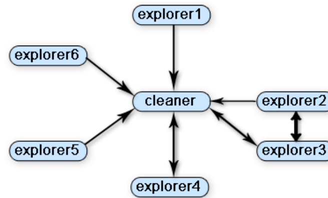


Fig. 1. The dynamic agent communication tool.

Another visualization tool is the static group tool. This debugging tool, which shows specific agent groups, is illustrated in figure 2. The line between the groups indicates the (amount) of communication overhead between the groups. In addition the developer can "jump into" a group and graphically view the agents and the communication between them.

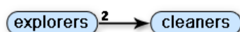


Fig. 2. The static group tool.

The static group tool can be helpful to quickly check if the correct agents are in the correct group. It can also be used to check communication between different groups. If two groups show an unusual amount of communication overhead the developer can jump into the group and locate the source of the problem. The marker to activate the static group tool can be specified as follow:

```
MADT(static_group_madt, @[explorers])
MADT(static_group_madt, @[cleaners])
```

The above markers update the tool on every change of the multi-agent program trace. According to these markers, the groups "explorers" and "cleaners" will be visualized. Generally it is most valuable to have a visualization of all communication between agents. However, to pinpoint the exact problem in a communication protocol it can be an invaluable addition to use a condition, which filters the messages that are shown. These same principles apply to the filtered view. As discussed in the related works section, other approaches (e.g., [4]) offers similar tools.

5 Conclusion

In this paper, we briefly discussed existing debugging approaches for multi-agent programs and presented a generic approach for debugging BDI-based multi-agent programs. Our approach is generic as it does not assume any specific representation for the internals of individual agents as well as the content of their exchanged messages. The proposed approach is based on an assertion language to express cognitive and temporal properties of the executions of multi-agent programs. The expressions of the assertion language can be used to trigger debugging tools such as breakpoints, watches, probes, and different visualization tools to examine and debug communication between individual agents. Since the assertion language is abstract, it can be applied to arbitrary BDI-based multi-agent programming languages.

We have already applied this debugging approach to 2APL [7] platform by modifying its corresponding interpreter to process debugging markers in a continuous mode. The 2APL interpreter evaluates the expressions of the assertion language based on the partial execution trace of the multi-agent programs. We have also implemented the proposed debugging tools that are discussed in this paper for the 2APL platform. The parser of 2APL is modified to analyze the markers included in the multi-agent program file. This implementation of 2APL is the official 2APL distribution that can be downloaded from `http://www.cs.uu.nl/2apl/`. It should be noted that the examples

discussed in this paper are already implemented in 2APL and the provided analysis is based on our implementation results.

We plan to extend the debugging mechanism of 2APL implementation such that debugging markers can be processed both in continuous and post mortem modes. Moreover, we believe that testing is an indispensable part of evaluating multi-agent programs and plan to follow the existing approaches on testing multi-agent programs and integrate them in our proposed debugging approach. In this way, we hope to generate a set of critical test traces and start debugging them in post mortem mode. Finally, in this proposal the groups are defined at design time and remain unchanged during the whole lifetime of the multi-agent program. A valuable extension is the ability to create dynamic groups, which defines the member of a group based on the cognitive state of one or more agents. One approach is to use a MDL assertion to specify which agents are members of the group. When the MDL assertion evaluates to true, the agent is a member of the defined group. To declare which agent would be a member of the group, a reserved agent variable name should be used. For example, every agent that can be substituted for the variable "MemberAgent" is a member of the defined group, if the MDL assertion evaluates to true.

References

1. F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi. JADE - a java agent development framework. In *Multi-Agent Programming: Languages, Platforms and Applications*. Kluwer, 2005.
2. R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni. *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, Berlin, 2005.
3. R.H. Bordini, M. Wooldridge, and J.F. Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons, 2007.
4. Juan A. Botía, Juan Manuel Hernansaez, and Antonio F. Gómez-Skarmeta. On the application of clustering techniques to support debugging large-scale multi-agent systems. In *PROMAS*, pages 217–227, 2006.
5. R. Collier. Debugging agents in agent factory. *ProMAS 2006*, pages 229–248, 2007.
6. K. S. Barber D. N. Lam. Debugging agent behavior in an implemented agent system. *ProMAS 2004*, pages 104–125, 2005.
7. Mehdi Dastani. 2APL: A practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.
8. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. In *Multi-Agent Programming: Languages, Platforms and Applications*. Kluwer, 2005.
9. David Poutakidis, Lin Padgham, and Michael Winikoff. Debugging multi-agent systems using design artifacts: The case of interaction protocols. In *In Proceedings of AAMAS-02*, pages 960–967, 2002.
10. David Poutakidis, Lin Padgham, and Michael Winikoff. An exploration of bugs and debugging in multi-agent systems. In *In Proceedings of the 14th International Symposium on Methodologies for Intelligent Systems (ISMIS)*, pages 628–632. ACM Press, 2003.
11. J. Sudeikat, L. Braubach, A. Pokahr, W. Lamersdorf, and W. Renz. Validation of BDI agents. *ProMAS 2006*, pages 185–200, 2007.
12. G. Vigueras and J. A. Botía. Tracking causality by visualization of multi-agent interactions using causality graphs. *ProMAS 2007*, pages 190–204, 2008.
13. M. Winikoff. JACK^{TM} intelligent agents: An industrial strength platform. In *Multi-Agent Programming: Languages, Platforms and Applications*. Kluwer, 2005.

Space-Time Diagram Generation for Profiling Multi Agent Systems

Author 1, Author 2, and Author 3

Institute Line 1
Institute Line 2
{author1,author2,author3}@example.com

Abstract. Advances in Agent Oriented Software Engineering have focused on the provision of frameworks and toolkits to aid in the creation of Multi Agent Systems. However, despite the inherent complexity of such systems, little progress has been made in the development of tools to allow for the debugging and understanding of their inner workings. This paper introduces a novel performance analysis system, named AgentSpotter, that is aimed at facilitating such analysis. AgentSpotter was developed by mapping more conventional profiling concepts to the domain of multi agent systems. We outline its integration into the Agent Factory multi agent toolkit.

1 Introduction

Recent developments in the area of Multi Agent Systems (MASs) have been concerned with bridging the gap between theory and practice, by allowing concrete implementations of theoretical foundations to be built and deployed. However, the dearth of agent-specific development and debugging tools remain a significant obstacle to MASs being adopted in industry on a large scale.

While some simple debugging and logging tools exist for MAS analysis, these tend not to aid in reasoning about large-scale system when viewed at the high agent-oriented abstraction layer. Such tools typically allow for traditional debugging actions such as state stepping and breakpoint insertion.

One popular performance analysis technique is known as *profiling*. Profiling is based on the observation that the majority of the execution time of a program can be attributed to a small number of *bottlenecks* (or *hot spots*). By improving the efficiency of these portions of a program, overall performance can be dramatically improved. Profiling was initially introduced by Donald E. Knuth in an empirical study conducted on FORTRAN programs [1]. Since then, the technique has been successfully applied to a variety of languages, platforms and architectures.

The aim of this paper is to apply the principles of traditional profiling systems in a multi agent environment, so as to facilitate the developers of MASs in debugging their applications by gaining a better understanding of where the bottlenecks exist and performance penalties are incurred.

This paper is organised as follows: Section 2 provides a brief overview of existing tools aimed at aiding in the analysis of MASs. In Section 3, we introduce

the AgentSpotter profiling system, with particular focus on outlining a conceptual model for generic MAS profiling. A concrete implementation of this work, aimed at the Agent Factory MAS framework is outlined in Section 4. Section 5 presents the agent call graph produced by AgentSpotter in more detail, with the evaluation of its usefulness given in Section 6. Finally, Section 7 presents our conclusions and ideas for future work.

2 Related Work

In designing a profiling application for MASs, it is necessary to identify the features that tend to be present in traditional profilers for non-MAS applications. It is also necessary to examine those debugging and analysis tools that already exist for MASs.

The motivation behind the use of profiling on computer applications is clearly outlined in Knuth's observation that "less than 4% of a program accounts for more than half of its running time" [1]. This statement implies that a developer can achieve substantial increases in performance by identifying and improving those parts of the program that accounts for the majority of the execution time. The key aim of profilers is to identify these bottlenecks.

Another observation leading to the widespread adoption of profilers as debugging tools is that there frequently exists a mismatch between the actual run-time behaviour of a system and the programmers' mental map of what they expect the behaviour to be. Profilers are useful in enlightening developers to particular aspects of their programs that they may not otherwise have considered.

A traditional profiler typically consists of two logical parts:

- An *instrumentation apparatus* that is directly weaved into the program under study or run side-by-side to gather and record execution data
- A *post-processing system* that uses the recorded data to generate meaningful performance analysis listings or visualisations

In the traditional software engineering community, historical profilers such as gprof [2] or performance analysis APIs like ATOM [3] and the Java Virtual Machine Tool Interface (JVMTI) [4] have made performance analysis more accessible for researchers and software engineers. However, the MAS community does not yet have general access to these types of tools.

Unique amongst all of the mainstream MAS development platforms, Cougaar is the only one that integrates a performance measurement infrastructure directly into the system architecture [5]. Although this is not applicable to other platforms, it does provide a good insight into the features that MAS developers could reasonably expect from any performance measurement application. The principal characteristics of this structure are as follows:

- Primary data channels consist of raw polling sensors at the heart of the system execution engine gather simple low-impact data elements such as counters and event sensors. These are triggered whenever the system steps through a predefined portion of the code.

– Secondary channels provide more elaborate information, such as summaries of the state of individual components and history analysis that stores performance data over lengthy running times.
– Computer-level metrics provide data on such items as CPU load, network load and memory usage.
– The message transport service captures statistics on messages flowing through it.
– An extension mechanism based on servlets allows the addition of visualisation plugins that bind to the performance metrics data source.
– The service that is charged with gathering these metrics is designed so as to have no impact on system performance when not in use.

Other analysis tools exist for aiding the development of MASs. However, these tend to be narrower in their focus, concentrating only on specific aspects of debugging MASs. The Agent Factory Debugger [6] is an example of a tool that is typical of most multi agent frameworks. Its principal function is inspecting the status and mental state of individual agents: its goals, beliefs, commitments and the messages it has exchanged with other agents. Tools such as this give limited information about the interaction between agents and the consequences of these interactions.

Another type of agent debugging tool is the ACLAnalyzer that has been developed for the JADE platform [7]. Rather than concentrating on individual agents, it is intended to analyse agent interaction in order to see how the community of agents interacts and is organised. In addition to visualising the number and size of messages sent between specific agents, it also employs clustering in order to identify cliques in the agent community.

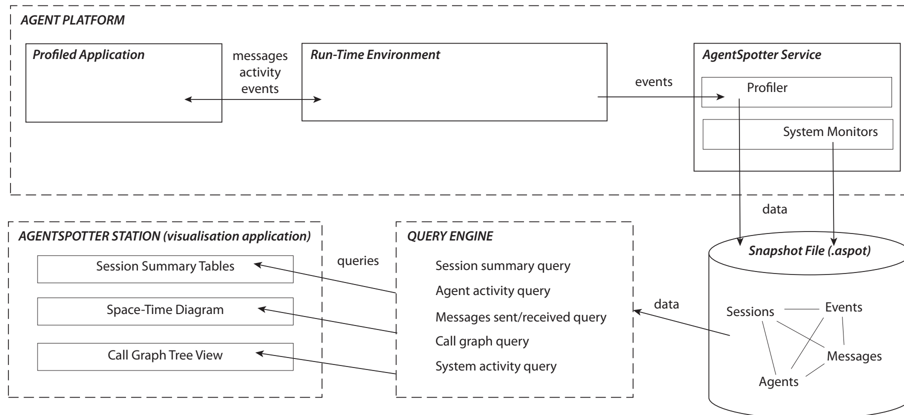
These latter tools are focused mostly on identifying what actions an agent is carrying out, together with identifying the reasons why such actions are taken (in response to the agents own belief set or as a result of receiving communication from other agents).

3 AgentSpotter Overview

The overriding objective of AgentSpotter is to map the traditional concepts of profiling to agent-oriented concepts so as to build a profiler tool for MAS developers. It could be argued that most mainstream agent toolkits are written in Java and so the existing profiling tools for the Java programming language are appropriate for the analysis of such platforms and their agents. However, to do so would necessitate the mapping of low-level method profiles to high-level agent-specific behaviour. Thus, tools aimed specifically at Java operate at an inappropriate conceptual level to be of use in agent analysis.

Ideally, MASs should be capable of managing their own performance and identifying their own bottlenecks that hamper system efficiency, and indeed much work is being undertaken towards this goal [8]. However, until this aim is realised, the provision of analysis tools aimed at aiming human developers identify issues with their systems remains of paramount importance.

Fig. 1. AgentSpotter abstract architecture



This section outlines the abstract infrastructure of the AgentSpotter system, that is capable of being integrated into any agent platform. Analysis of the integration of AgentSpotter into a specific agent platform (namely Agent Factory) is contained in Section 4.

The AgentSpotter abstract architecture is displayed in Figure 1, using the following graphical conventions:

– Top-level architectural units are enclosed in dashed lines and are titled in slanted capital letters, e.g. ***AGENT PLATFORM***
– Self-contained software packages are enclosed in solid lines e.g Profiler
– Logical software modules (groups of packages) are titled using slanted capitalised names e.g. ***AgentSpotter Service***
– Arrows denote data or processing interactions e.g. queries

At the highest level, the *AgentSpotter Service* should communicate with the *Run-Time Environment* to capture the profiling data from a *Profiled Application* running inside an *Agent Platform*. The captured data should be stored into a *Snapshot File* which would then be processed by a *Query Engine* to generate the input data for *AgentSpotter Station*, the visualisation application.

The AgentSpotter profiler monitors performance events generated by the Agent Platform's Run-Time Environment. These include such events as agent management events, agent scheduler activity, messaging other platform service activity. Additionally, the AgentSpotter service may employ system monitors to record performance information such as CPU load, memory usage or network throughput. This provides a general context for the event-based information being gleaned by the profiler.

The event data and other information collected by the AgentSpotter profile is stored in a *snapshot file*, which contains the results of a single uninterrupted data capture session. This snapshot contains a series of raw instrumentation data. A difficulty arises in that a large MAS may generate potentially hundreds of events per second. For this reason, it is necessary to introduce a *Query Engine* that is capable of extracting summaries and other information and make it available to visualisation tools in a transparent manner. Ideally, this should be through a data manipulation language such as SQL so as to facilitate the specification of rich and complex queries.

The final component of the abstract architecture is the *AgentSpotter Station*, which is the visualisation tool that summarises the information gathered from the Query Engine in a visual form. The principal focus of this paper is the Space-Time Diagram, which is presented in Section 5.

At this stage of planning an agent profiler, it is vital to decide upon the minimum information that should be available from the system. When profiling any application, it is important to identify the appropriate execution unit for profiling (e.g. in the context of object-oriented programming, this would typically be an object or a method). For profiling a MAS, we believe that the appropriate execution units are individual agents, in which case the information provided should include:

- **Agent description:** name, role, type (deliberative or reactive) of the agent.
- **Cumulative activity:** cumulative computation time used by the agent.
- **Perception time:** perception time used by a deliberative agent.
- **Action time:** task execution time used by a deliberative agent.
- **Reasoning time:** reasoning time used by a deliberative agent.
- **% session activity** percentage of the global session computation time used by the agent.
- **Number of iterations:** number of non-zero duration iterations used by the agent.
- **Number of time slice overshoots:** number of times where an agent has overused its time slice allocation.
- **Maximum and average iteration duration:** maximum and average duration of these time slices
- **Total number of messages sent and received:** total number of ACL messages exchanged by the agent.

In addition to these agent-specific metrics, a number of global statistics should also be maintained:

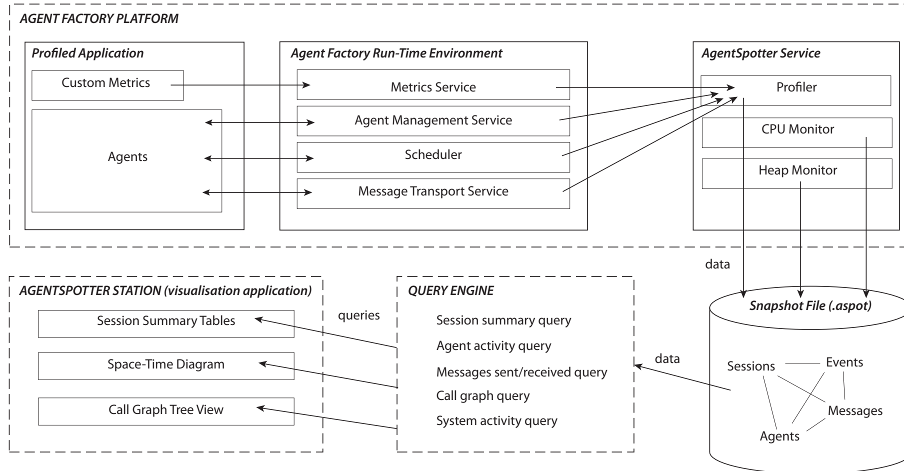
- **Total duration:** session run-time recorded.
- **Total activity:** amount of computation time recorded over the session.
- **Total number of messages:** number of messages sent or received by agents on the platform being profiled.
- **Average number of active agents per second:** This gives an idea of the level of concurrency in the application.

Following the convention of traditional profiling tools, we describe this information as a *flat profile*. AgentSpotter displays this by means of a JTable (provided by Java's Swing interface tools). An example of how the information is presented is given below in Table 1.

4 Agent Factory Integration

Following the definition of the abstract architecture outlined above, a concrete (i.e. platform-specific) implementation was created for Agent Factory. Agent Factory is a cohesive framework that supports a structured approach to the development of agent-oriented applications [9]. This implementation is illustrated in Figure 2, which uses the same graphical conventions as Figure 1.

Fig. 2. AgentSpotter concrete architecture



In order to create a concrete implementation, only the platform-specific details must change, as the mechanisms required to monitor events varies from one agent platform to another. In contrast, the AgentSpotter file processing and visualisation components (shown in the lower part of Figure 2) are identical to those in the abstract architecture (Figure 1). Thus, when making use of AgentSpotter for a new type of agent platform, only the AgentSpotter Service that is coupled directly with the platform needs to be reprogrammed. Provided this service creates snapshot files in a consistent way, the Query Engine need not differentiate between agent platforms.

Within, the Agent Factory Run-Time Environment, there are three specific subsystems that generate events that are of interest in agent profiling, and as

such are recorded by the AgentSpotter service. First, the *Agent Management Service* is responsible for creating, destroying, starting, stopping and suspending agents. It generates events corresponding to each of these actions, which are recorded in the snapshot file. The *Scheduler* is charged with scheduling which agents are permitted to execute at particular times and generates events based on this. Finally, the *Message Transport Service* records the sending and receipt of FIPA messages by agents.

5 Space-Time Diagram

In Section 3, we outlined the minimum amount of information that should be made available by an agent profiler. However, this information can be presented merely by the creation of a simple table. We believe that proper visualisation tools will be far more useful to a developer in understanding a MAS. This section introduces the Space-Time Diagram that is at the core of the AgentSpotter Station visualisation application. The aim of this diagram is to make as much detail and context about the performance of the MAS available to the developer. The user may pan the view around and zoom in and out so as to reveal hidden details or focus on minute details.

The *Session Time Line* represents the running time of the application being profiled. Regardless of the position and scale of the current viewport, this time line remains visible to provide temporal context to the section being viewed and also to allow a developer to move to various points in the session.

The *CPU Line* displays a graphical of the CPU load of the host system during the session. A vertical gradient going from green (low CPU usage) to red (high CPU usage) provides a quick graphical sense of system load. A popup information window reveals the exact usage statistics once the mouse is hovered over the line.

Perhaps the most important feature of the space-time diagram is the *Agent Time Lines*. Each of these display all the performance and communication events that occur for a single agent during a profiling session. A number of visual features are available to the developer so as to gain greater understanding of the status and performance of the system. For instance, an agent time line begins only at the point in time when the agent is created. This facilitates the developer in viewing the fluctuations in the agent population. Another simple visual aid is that a life line's caption (i.e. the name of the associated agent) is always visible, regardless of what position along the line a developer has scrolled to. Visual clutter may also be reduced by temporarily hiding certain life lines that are not of interest at a particular point in time.

The time line also changes colour in order to distinguish busier agents from the remainder of the community. Darker lines indicate agents that have consumed a greater proportion of the system's CPU time. In a situation where system performance has been poor, this will allow a developer to quickly identify candidate agents for debugging, if they are consuming more resources that is appropriate or expected.

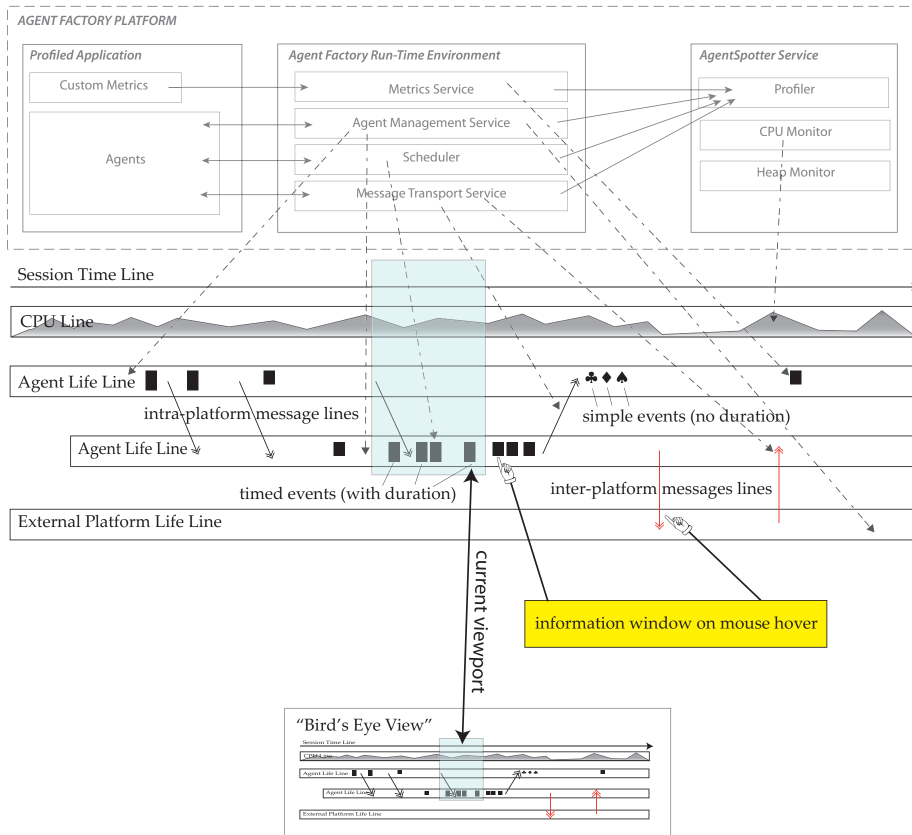


Fig. 3. AgentSpotter Space-Time Diagram specification annotated with AgentSpotter for Agent Factory infrastructure links (see Figure 2).

The default ordering of the time lines shares this aim. The ordering of the time lines is in descending order of total computation time, again visually notifying the developer of those agents consuming more processing resources. However, a developer may alter this default order by dragging time lines into different positions, perhaps to group life lines with particularly interesting interactions.

In addition to this simple information, the main purpose of the time line is to show events performed by an agent that is likely to be of interest from a performance point of view. These *performance events* are divided into two categories. *Simple performance events* are those that have a time stamp only. These are shown by means of standard icons (such as an envelope icon to denote that a message was received by the agent).

The other category of performance events are *timed performance events*. These events are typically actions being performed by an agent. An example of how these timed performance events are represented is given in Figure 4. Each agent has a particular timeslice within which it is expected to perform all of its actions

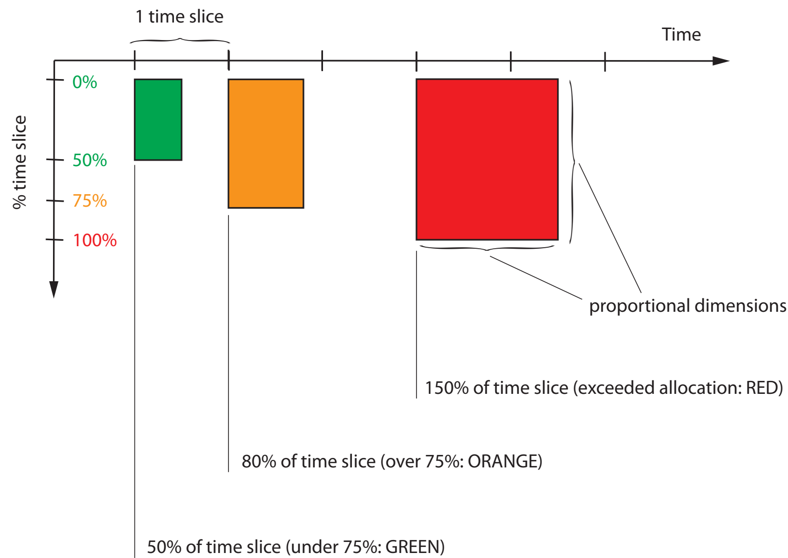


Fig. 4. Agent Factory agent activity representation in the Space-Time Diagram

in a particular iteration. Agents exceeding their time slice may prevent or delay other agents from getting access to the CPU. As a visual aid to identifying when this situation occurs, timing events are represented by coloured rectangles. The size of these rectangles is proportional to the duration of the event and the percentage of the allocated timeslice used. The colour code also indicates how the agent has used its time available. A green rectangle indicates that the agent has used anything up to 75% of the time available. From 75% to 100%, an orange rectangle indicates that the agent has used most of its allocated time, and may require further analysis from the developer to avoid the danger of exceeding the time slice. Finally, whenever an agent exceeds its time, a red rectangle is used.

For either form of event, hovering the mouse over the event indicator (whether an icon for a simple event or a rectangle for timed events) will cause a popup window to display the specific details about the event.

Communication between agents is shown by means of lines linking the appropriate agent timelines. These have arrows attached so as to clearly indicate the direction of the communication. Hovering the mouse pointer over such a line causes a popup information window to display the FIPA headers and content of the message that was sent. There is also a distinction made between messages passed between agents housed on the same agent platform (intra-platform) and those passed between agents on different platforms (inter-platform). Since agents on other platforms will not have an agent time line associated with them, an *external platform life line* is drawn for each other platform with which agents on the platform being profiled communicate. Rather than linking with individual

agent time lines, communications with these platforms are drawn directly to the external platform life line.

The combination of these communication lines and the performance event indicators are very useful in identifying the causes of agent activity. Given the inherently social nature of MASs, it is very common for agent activity to be motivated by communication in some way. For example, an agent may be requested to perform a task by some other agent. Alternatively, an agent may receive a piece of information from another agent that it requires in order to perform a task that to which it has previously committed as a result of its own goals and plans.

Providing such a detailed visualisation of a MAS requires a substantial amount of screen space. The basic features of zooming and panning are complimented by the provision of a "bird's eye view", which displays a zoomed-out overview of the entire session. This allows the user to quickly move the current viewport to focus on a particular point in time during the session, as illustrated in Figure 3.

6 Evaluation

Having outlined the required features of AgentSpotter, along with details of its implementation, it is necessary to demonstrate how it can be utilised on a running MAS. To this end, a specialist benchmark application was developed that will allow the features of the AgentSpotter application to be shown.

6.1 Specification

The aim of the benchmark application is to perform all the activities necessary for AgentSpotter to display its features. The requirements for the application can be summarised as follows:

- **Load history:** a normally distributed random load history should be generated so that we can get an idea of a "normal" profile which can be contrasted with "abnormal" profiles where, for example, a single agent is monopolising all the load, or the load is spread equally among all agents.
- **Agent population:** the number of active agents should be changeable dynamically to simulate process escalation.
- **Interactions:** in addition to direct interactions, the application should exercise some task delegation scenarios. The idea is to generate multiple hops messaging scenarios and see their impact on performance.
- **Messages:** agents should generate a steady flow of messages with occasional bursts of intense communication.
- **Performance events:** all three performance behaviours described in 3 should be represented, i.e. green ($t \leq 50\%$ time slice), orange ($50\% \leq t \leq 75\%$ time slice), and red ($t > 100\%$).

These requirements were satisfied by creating a MAS with overseer agents that request worker agents to execute small, medium or large tasks. Worker

agents that have been recently overloaded will simply refuse to carry out the tasks (in a real application they would inform back the requester about their refusal). From time to time, overseer agents would request agents to delegate some tasks. In this case, worker agents will behave as overseers just for one round. A simple interface allows the user to start and pause the process, along with the ability to set the number of active worker agents.

6.2 Evaluation Scenario and Objective

The following simple scenario was played out in order to gather the information required for the AgentSpotter application to generate its flat profile and space-time diagram.

1. Start the session with 12 worker agents and 2 overseer agents.
2. After 10 minutes add 15 worker agents to spread the load.
3. After 4 further minutes, suspend the process for 20 seconds.
4. At this point, reduce the number of worker agents to a 12.
5. Run for 5 minutes more then stop the session.

6.3 Flat profile

The resulting flat profile of this test is reproduced in Table 1. For the reader's convenience, the maximum value for each column is identified by an enclosing box. Overseer agents are called "master1" and "master2". The worker agents are called "agent" followed by a number e.g. "agent007".

Firstly, the benchmark appears to make a good job of producing a load history following a normal distribution.

Secondly, we can draw the following conclusions from a quick study of Table 1:

- The most active agents in terms of number of iterations are the overseer agents, "master1" and "master2", however in terms of CPU load and overload, three worker agents are topping the list with 30% of the total activity: "agent001", "agent009", and "agent003".
- The agents with the highest CPU load also display a high number of time slice overshoots, and a high average time slice duration.
- As expected, the overseer agents were very busy exchanging messages with the workers. However, it seems that messaging is not CPU intensive. This is possibly as a result of the way in which message sending is implemented, with the CPU load indicated here corresponding to the scheduling of a message for sending, rather than the actual sending of the message. It may be necessary to attach a specialist monitor to the Message Transport Service to gain full information about the impact of sending messages. This causes the activity percentage of the overseer agents to be very low, at only 1%.

In this instance, the flat profile lends evidence to the notion that the actual behaviour of the system matches the design principles on which it was built.

Table 1. Benchmark application flat profile

Total Session Time	18:50.691
Total Activity	10:29.164
Messages Sent	1206
Messages Received	1206
Time Slice Duration	1000 ms

Agent	$T > 0$ iterations	$T > 100\%$ overload	Activity mm:ss.ms	% Session activity	$Max(T)$ ss.ms	$Average(T)$ ss.ms	Msg. sent	Msg. rec.
agent001	338	22	1:08.564	10.90	3.740	0.202	6	57
agent009	365	21	1:04.257	10.21	3.425	0.176	13	77
agent004	349	22	1:01.529	9.78	3.235	0.176	10	69
agent014	284	14	46.413	7.38	3.148	0.163	2	36
agent003	401	13	43.881	6.97	3.323	0.109	12	76
agent006	361	12	40.141	6.38	3.279	0.111	12	73
agent005	367	12	34.903	5.55	3.325	0.095	17	76
agent013	301	9	34.716	5.52	3.190	0.115	14	71
agent007	378	11	31.864	5.06	3.356	0.084	21	71
agent008	357	7	30.850	4.90	3.201	0.086	14	72
agent010	330	8	30.280	4.81	3.147	0.091	21	81
agent015	285	9	29.382	4.67	3.257	0.103	4	42
agent002	348	8	23.196	3.69	3.147	0.066	9	70
agent011	357	5	19.363	3.08	3.095	0.054	4	39
agent012	225	3	13.172	2.09	3.049	0.058	9	41
master2	901	0	6.681	1.06	0.183	0.007	504	86
master1	873	0	6.485	1.03	0.227	0.007	514	82
agent024	46	2	6.281	1.00	3.045	0.136	3	7
agent019	31	1	4.449	0.71	3.014	0.143	0	5
agent026	42	1	4.400	0.70	3.084	0.104	0	4
agent030	26	1	4.002	0.64	3.132	0.153	2	8
agent017	46	1	3.811	0.61	3.031	0.082	0	3
agent025	40	1	3.767	0.60	3.006	0.094	0	3
agent027	31	1	3.694	0.59	3.103	0.119	0	2
agent018	38	1	3.384	0.54	3.044	0.089	2	7
agent020	39	0	1.762	0.28	0.547	0.045	0	3
agent022	47	0	1.523	0.24	0.559	0.032	5	13
agent021	32	0	1.300	0.21	0.555	0.040	2	7
agent016	219	0	1.194	0.19	0.555	0.005	2	6
agent029	38	0	1.039	0.17	0.550	0.027	2	8
agent028	45	0	0.749	0.12	0.546	0.016	1	4
agent032	34	0	0.749	0.12	0.561	0.022	1	4
agent031	36	0	0.742	0.12	0.545	0.020	0	2
agent023	40	0	0.598	0.10	0.543	0.014	0	1
agent033	30	0	0.043	0.01	0.003	0.001	0	0

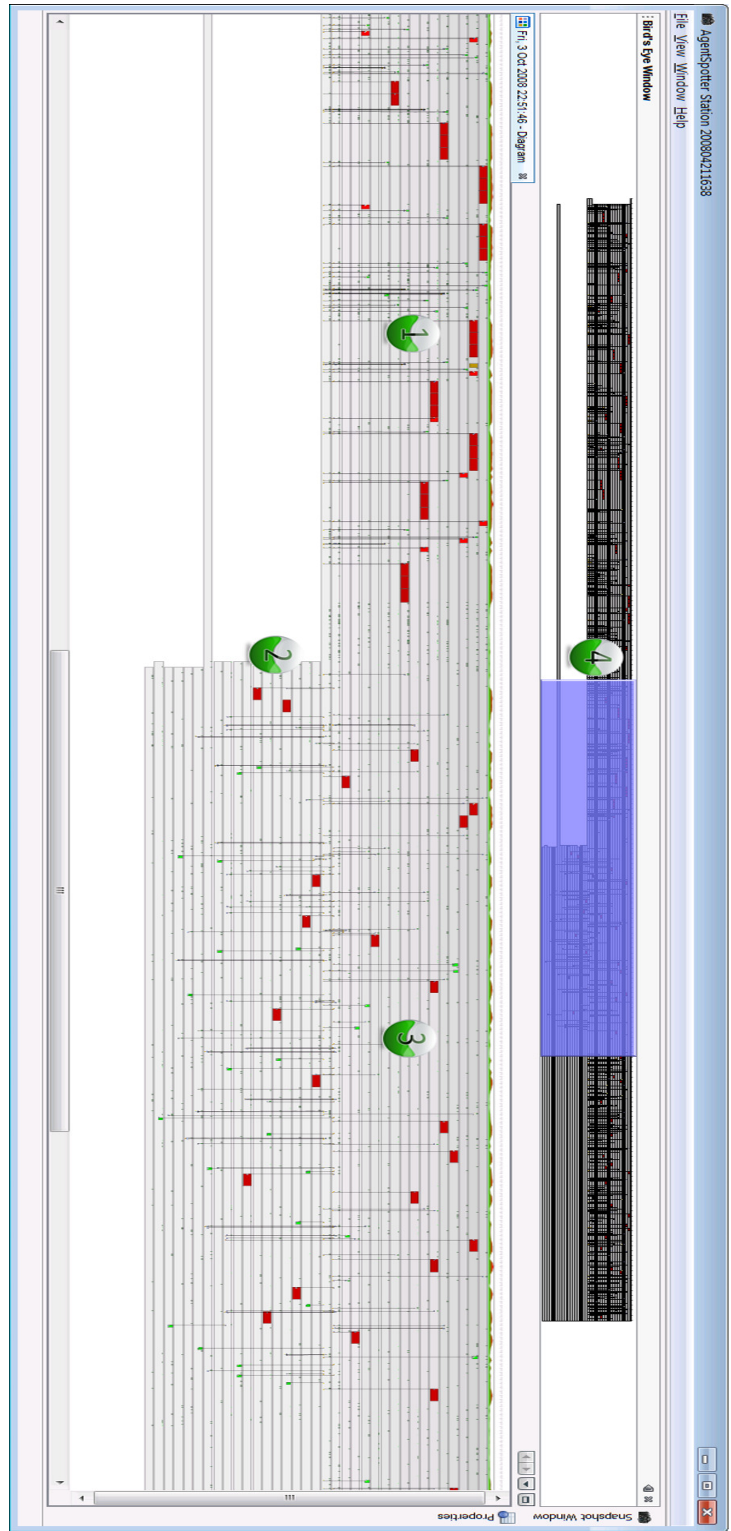


Fig. 5. Benchmark application sample space-time diagram (18 minute long session)

6.4 Space-Time Diagram

The space-time diagram associated with this session is displayed in Figure 5. In this diagram, the individual agent time lines can clearly be seen as horizontal bars in the main window of the application. Within these, rectangular boxes represent processing tasks being carried out by each agent. The vertical lines between the time lines represent messages being passed between agents within the system. For this simple scenario, only a single agent platform was used, meaning that there are no external platform life lines to indicate messages travelling to and from other agent platforms. A number of points of interest are labelled on the diagram. These can be described as follows:

1. This portion of the diagram shows what happens when the initial 12 workers are active. The large red rectangles illustrate the time-consuming tasks ordered by the overseer agents. As mentioned in Section 5, these are also identifiable by their size, which increases proportionally to the processing time taken. These blocks never overlap because of the way Agent Factory schedules agents (i.e. agents are given access to the CPU sequentially, rather than each agent running in a concurrent thread of execution). It is also noteworthy that the Agent Factory scheduler does not preempt agents that have exceeded their time allocation. The red rectangles also come in bursts, because both overseers send the same order to the same worker at the same time. This was revealed by zooming into what initially appeared to be a single message line. At a high magnification level, there were in fact two messages lines within a few microseconds interval to the same worker.
2. At this point, 15 more workers are added to the system, following a slight pause that is indicated by the temporary absence of message lines. The agent time lines for the additional agents only begin at this point, clearly indicating an increase in the agent population.
3. This third portion shows the impact of the new workers. The red blocks are still present, but they are better spread among the agents, with the new agents taking some of the load from their predecessors.
4. The Bird's Eye View reveals the bigger picture, and reminds us that we are looking only at one third of the overall session.

7 Conclusions and Future Work

Currently, the only concrete implementation of AgentSpotter is for the Agent Factory platform. As noted in Section 3, only the data capture apparatus should require a separate implementation for another platform. It is intended to develop such an implementation for other platforms, such as JADE [10].

The most obvious source of improvement for the AgentSpotter application is the addition of extra information above that which is already available. For instance, the performance of additional system services should be recorded, and more details should be collected about agents' performance events, such as the distribution of an agent's execution time among its sensors, actuators, reasoning

engine and other components. Finally, the AgentSpotter application currently supports only one agent platform at any given time. The capability to visualise multiple platforms concurrently would be desirable.

References

1. Knuth, D.E.: An empirical study of FORTRAN programs. j-SPE **1**(2) (April/June 1971) 105–133
2. Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: A call graph execution profiler. SIGPLAN Not. **17**(6) (1982) 120–126
3. Srivastava, A., Eustace, A.: Atom: a system for building customized program analysis tools. In: PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, New York, NY, USA, ACM (1994) 196–205
4. Sun Microsystems, Inc.: JVM Tool Interface (JVMTI), Version 1.0. Web pages at `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/` (accessed August 4th, 2008) (2004)
5. Helsinger, A., Thome, M., Wright, T., Technol, B., Cambridge, M.: Cougaar: a scalable, distributed multi-agent architecture. In: Systems, Man and Cybernetics, 2004 IEEE International Conference on. Volume 2. (2004)
6. Collier, R.: Debugging Agents in Agent Factory. Lecture Notes in Computer Science **4411** (2007) 229
7. Botia, J., Hernansaez, J., Skarmeta, F.: Towards an Approach for Debugging MAS Through the Analysis of ACL Messages. In: Multiagent System Technologies: Second German Conference, MATES 2004, Erfurt, Germany, September 29-30, 2004: Proceedings, Springer (2004)
8. Horn, P.: Autonomic Computing: IBM's Perspective on the State of Information Technology. IBM TJ Watson Labs, NY, 15th October (2001)
9. Collier, R., O'Hare, G., Lowen, T., Rooney, C.: Beyond Prototyping in the Factory of Agents. Multi-Agent Systems and Application III: 3rd International Central and Eastern European Conference on Multi-Agent Systems, Ceemas 2003, Prague, Czech Republic, June 16-18, 2003: Proceedings (2003)
10. Bellifemine, F., Poggi, A., Rimassa, G.: JADE–A FIPA-compliant agent framework. In: Proceedings of PAAM. Volume 99. (1999) 97–108

An Open Architecture for Service-Oriented Virtual Organizations

A. Giret, V. Julián, M. Rebollo, E. Argente, C. Carrascosa, V. Botti

Dept. Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n – 46022 Valencia – Spain
{agiret, vinglada, mrebollo, eargente, carrasco, vbotti}@dsic.upv.es

Abstract. Recent technological advances in open systems have imposed new needs on multi-agent systems. Nowadays, open systems require open autonomous scenarios in which heterogeneous entities (agents or services) interact to fulfill the system goals. The main contribution of this paper is the definition of an open architecture and computational model for large-scale open multi-agent systems based on a service-oriented approach. The new proposed architecture, called THOMAS, is specifically addressed for the design of virtual organizations. A simplified example for the management of a travel agency system, which shows the features of the proposal, is also included.

1 Introduction

The technological advances of recent years have defined the "new society", in which a multi-agent system participates as an open environment in which heterogeneous entities (agents and services) interact. This new environment needs to meet several requirements such as: distribution, constant evolution, flexibility to allow members enter or exit the society, appropriate management of the organizational structure that defines the society, multi-device agent execution including devices with limited resources, and so on. All these requirements define a set of features that can be addressed through the open system paradigm and virtual organizations.

Regarding organizations, this paradigm has been conceived as an encouraging solution for managing coordination and controlling agent behavior, specially in open multi-agent systems [11]. Organization modeling not only allows describing structural composition (i.e. roles, agent groups, interaction patterns, role relationships) and functional behavior (i.e. agent tasks, plans or services), but also normative regulations for controlling agent behavior, dynamic entry/exit of components and dynamic formation of agent groups.

Over recent years, several works have appeared trying to solve the problem of integrating the multi-agent system paradigm and the service-oriented computing paradigm. By integrating these two technologies it is possible to model autonomous and heterogeneous computational entities in dynamic and open environments. Such entities may be reactive, proactive and have the ability to

communicate in a flexible way with other entities [27]. The *Agent and Web Services Interoperability* (AWSI) IEEE FIPA Working Group [1] proposes to create links, as a gateway, between the two approaches. In contrast, we propose a new open multi-agent system architecture consisting of a related set of modules that are suitable for the development of systems applied in environments such as those raised above. This new architecture is called THOMAS (Me*TH*ods, Techniques and Tools for *O*pen *M*ulti-*A*gent *S*ystems). The proposed solution tries to communicate agents and web services in a transparent, but independent, way, going beyond related works, raising a total integration of both technologies. So agents can offer and invoke services in a transparent way to other agents or entities, as well as external entities can interact with THOMAS agents through the use of the offered services.

This paper is structured as follows: Section 2 presents the proposed architecture model. The description of the services offered by the THOMAS main components are described in Sections 3 and 4. Section 5 shows a simplified example of a travel agency in which the overall functioning of the THOMAS architecture can be observed. In Section 6, a prototype of THOMAS is overviewed. Section 7 presents a discussion on the features of the proposal related with state of the art works. Finally, conclusions are presented.

2 THOMAS Architecture

THOMAS architecture basically consists of a set of modular services. Though THOMAS feeds initially on the FIPA architecture, it expands its capabilities to deal with organizations, and to boost up its service abilities. In this way, a new module in charge of managing organizations has been introduced, along with a redefinition of the FIPA *Directory Facilitator* that is able to deal with services in a more elaborated way, following *Service Oriented Architectures*[2] guidelines. As it has been stated before, services are very important in THOMAS. In fact, agents have access to the THOMAS infrastructure through a range of services included on different modules or components. The main components of THOMAS are the following (Figure 1):

- *Service Facilitator* (SF), it offers simple and complex services to the active agents and organizations. Basically, its functionality is like a yellow page service and a service descriptor in charge of providing a green page service. The detailed description of this module is presented in Section 3.
- *Organization Management System* (OMS), it is mainly responsible of the management of the organizations and their entities. Thus, it allows creation and management of any organization. The OMS is described in Section 4.
- *Platform Kernel* (PK), it maintains basic management services for an agent platform. The PK represents any FIPA compliant platform. In this way, THOMAS can be configured to work with any agent platform which implements the FIPA AMS and the FIPA communication network layer.

[1] http://www.fipa.org/subgroups/AWSI-WG.html
[2] http://www.oasis-open.org

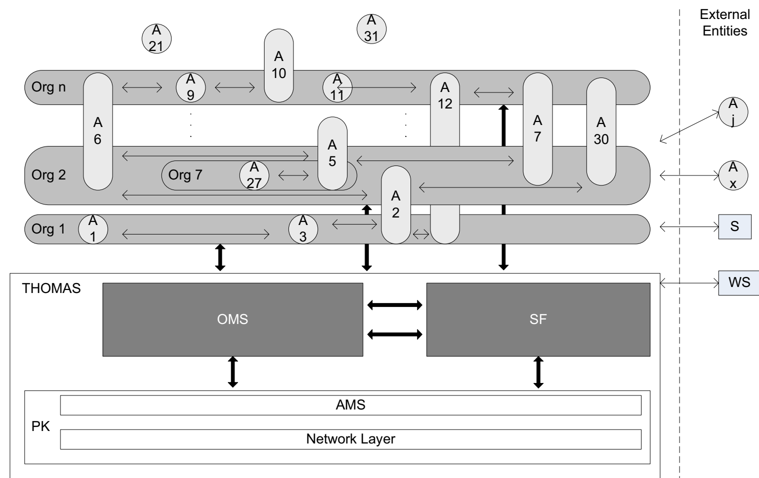


Fig. 1. THOMAS Architecture (Notation used: "Org" represents an organization; "A" an agent; "S" an external service; and "WS" a Web Service)

The following sections describe, in a deeper detail, the main components of the THOMAS architecture.

3 Service Facilitator

The Service Facilitator (SF) is a mechanism and support by which organizations and agents can offer and discover services. The SF provides a place in which the autonomous entities can register service descriptions as directory entries.

The SF acts as a gateway to access the THOMAS platform. It manages this access transparently, by means of security techniques and access rights management. The SF can find services searching for a given service profile or searching by the goals that can be fulfilled when executing the service. This is done using the matchmaking and service composition mechanisms that are provided by the SF. The SF also acts as a yellow pages manager and in this way it can find which entities provide a given service.

A service represents an interaction of two entities, which are modeled as communications among independent processes. Regarding the service description, keywords or semantic annotations can be used. Languages as OWL-S[17], WSMO[3], SAWSDL[8] or WSDL-S[4] are the most used ones to describe services.

Furthermore, a service offers some capabilities, each of which enables fulfilling a given goal. Services are characterized by their inputs, outputs, preconditions and effects. Furthermore, there could be additional parameters in a service

[3] http://www.wsmo.org/wsml/wsml-syntax
[4] http://www.w3.org/Submission/WSDL-S/

description, which are independent of the service functionality (non-functional parameters), such as quality of service, deadlines and security protocols. Taking into account that THOMAS works with semantic services, another important data is the ontology used in the service. Thus, when the service description is accessed, any entity will have all needed information in order to interact with the service and make an application that can use this service. Such a description can also be employed for pre-compiled services, in which the process model of the service is composed of the sequence of the elementary services that will be executed, instead of the internal processes of this service.

Therefore, OWL-S is the language chosen to represent semantic services in the SF. This language has been extended to empower its functionality, adding goals, preconditions and effects (or postconditions) as logical formulas.

A service can be supplied by more than one provider in the system. In this way, a service has an associated list of providers. All providers can offer exact copies of the service, that is, they share a common implementation of the service. Or they may share only the interface and each provider may implement the service in a different way. This is easily achieved in THOMAS because the general service profile is separated from the service process.

A service is defined as a tuple *<sID, goal, prof, proc, ground, ont>* where:

- sID is an unique service identifier.
- *goal* is the final purpose of the service, composed by a set of abstract concepts provided in the system's design. It gives a first abstraction level for service search and composition.
- $prof$ is the service profile that describes the service in terms of its IOPEs (Inputs, Outputs, Preconditions and Effects) and non-functional attributes, in a readable way for those agents that are searching information.
- *proc* specifies how to call a service and what happens when the service is executed.
- *ground* specifies in detail how an agent can access the service. A grounding specifies a communication protocol, the message formats, the contact port and other specific details of the service.
- *ont* is the ontology that gives meaning to all the elements of the service. OWL-DL is the chosen language.

The tuple defined above for service specification is implemented in two parts: the *abstract service*, general for all providers; and the *concrete service*, with the implementation details. In this way, services are stored inside the system split into these two parts: the *service profile* (that represents the abstract service specification) and a set of *service processes specifications* (that detail the concrete service). Thus, in THOMAS services are implemented as the following tuple:

<ServiceID, Providers, ServGoal, ServProfile>
Providers ::= <ProvIDList, ServImpID, ServProcess, ServGround> $^+$
ProvIDList ::= ProviderID$^+$

where:

– *Providers* is a set of tuples composed of a *Providers identifier list* (ProvIDList), the service process model specification (*ServProcess*), and its particular instantiation (*ServGround*).
– *ProvIDList* maintains a list of service provider identifiers.

The SF supplies a set of standard services to manage the services provided by organizations or individual agents. These services can also be used by the rest of THOMAS components to advertise their own services. SF services are classified in three types:

– *Registration*: they allow to add, modify and remove services from the SF directory.
– *Affordability*: for managing the association between providers and their services.
– *Discovery*: for searching and composing services as an answer to user requirements.

The complete relation of the SF services can be found in Table 1.

Service	**Description**
Registration	
RegisterProfile(?p:Profile, ?g:Goal)	Creates a new service description (*profile*)
RegisterProcess(?s:ID, ?pr:Process, ?gr:Grounding, ?prov:ID)	Creates a particular implementation (*process*) for a service
ModifyProfile(?s:ID, ?p:Profile, ?g:Goal)	Modifies an existing service profile
ModifyProcess(?sImp:ID, ?pr:Process, ?gr:Grounding)	Modifies an existing service process
DeregisterProfile(?s:ID)	Removes a service description
Affordability	
AddProvider(?sImp:ID, ?prov:ID)	Adds a new provider to an existing service process
RemoveProvider(?sImp:ID, ?prov:ID)	Removes a provider from a service process
Discovery	
SearchService(?pu:ServPurpose)	Searches a service (or a composition of services) that satisfies the user requirements
GetProfile(?s:ID)	Gets the description (profile) of a specific service
GetProcess(?sImp:ID)	Gets the implementation (process) of a specific service

Table 1. SF Services

4 Organization Management System

The *Organization Management System (OMS)* is in charge of the organization life-cycle management, including specification and administration of both the structural components of the organization (roles, units and norms) and its execution components (participant agents and roles they play).

Organizations are structured by means of *organizational units* (OUs), which represent groups of entities (agents or other units), that are related in order to pursue a common goal. These OUs have an internal structure (i.e. hierarchical, team, plain), which imposes restrictions on agent relationships and control

(ex. supervision or information relationships). OUs can also be seen as virtual meeting points because agents can dynamically enter and leave them by means of adopting (or leaving) roles inside. Roles represent all required functionality needed in order to achieve the unit goal. They might also have associated norms for controlling role actions. Agents can dynamically adopt roles inside units, so a control for role adoption is needed. Finally, services represent some functionality that agents offer to other entities, independently of the concrete agent that makes use of them.

The OMS keeps record on which are the Organizational Units of the system, the roles defined in each unit and their attributes, the entities participating inside each OU and the roles that they enact through time. Moreover, the OMS also stores which are the norms defined in the system. Regarding roles, the role attributes are: *accesibility*, that indicates whether a role can be adopted by an agent on demand; *visibility*, that indicates whether agents can obtain information from this role on demand; *position*, that indicates whether it is a supervisor, subordinate or simple member of the unit; and *inheritance*, that indicates which is its parent role, establising a hierarchy of roles.

The OMS offers a set of services for organization life-cycle management, classified in (Table 2): (i) structural services, which modify the structural and normative organization specification; (ii) informative services, that provide information of the current state of the organization; and (iii) dynamic (role-management) services, which allow managing dynamic entry/exit of agents and role adoption.

The **structural services** deal with adding/deleting norms (*RegisterNorm*, *DeregisterNorm*), adding/deleting roles (*RegisterRole*, *DeregisterRole*) and creating new organizational units or deleting them (*RegisterUnit*, *DeregisterUnit*).

The **informative services** give specific information of the current state of the organization, detailing which are the roles defined in an OU (*InformUnitRoles*), the roles played by an agent (*InformAgentRoles*), the specific members that participate inside an OU (*InformMembers*), the number of members of an OU (*InformQuantity*), its internal structure (*InformUnit*), and the services and norms related with a specific role (*InformRoleProfiles*, *InformRoleNorms*).

The **dynamic services** allow defining how agents can adopt roles inside OUs (*AcquireRole*, *LeaveRole*) or how agents can be forced to leave a specific role (*Expulse*), normally due to sanctions.

By means of the publication of the *structural services*, the OMS allows the modification of some aspects related to the organization structure, functionality or normativity at execution time. For example, a specific agent of the organization can be allowed to add new norms, roles or units during system execution. These types of services should be restricted to the internal roles of the system, which have a level of permission high enough to these kinds of operations (i.e. supervisor role). Moreover, these services might not be published in the SF in some specific applications in which the system structure must not be dynamically modified. The *information services* might also be restricted to some internal roles of the system, as they provide with specific information of all the components of the organization

Service	Description
Structural Services	
RegisterRole(?Role:ID, ?Unit:ID, ?Attr:Attributes)	Creates a new role within a unit, with specific attributes (visibility, accesibility, position, inheritance)
RegisterNorm(?Norm:ID, ?Role:ID, ?Content: NormContent, ?Issuer:ID, ?Defender:ID, ?Promoter:ID)	Includes a new norm within a unit, indicating its content (deontic value, conditions, actions and associated sanctions or rewards)
RegisterUnit(?Unit:ID, ?UnitType:Type, ?UnitGoal:Goal, [?UnitParent:ID])	Creates a new unit within a specific organization, indicating its structure (type), goal and its parent inside the organization hierarchy
DeregisterRole(?Role:ID, ?Unit:ID)	Removes a specific role description from a unit
DeregisterNorm(?Norm:ID)	Removes a specific norm description
DeregisterUnit(?Unit:ID, [?UnitParent:ID])	Removes a unit from an organization
Informative Services	
InformAgentRole(?Agent:ID)	Indicates roles adopted by an agent
InformMembers(?Unit:ID, [?Role:ID])	Indicates entities that are members of a specific unit. Optionally, indicates only members playing the specific role inside that unit.
QuantityMembers(?Unit:ID, [?Role:ID])	Provides the number of current members of a specific unit. Optionally, it indicates only the number of members playing the specific role inside the unit
InformUnit(?Unit:ID)	Provides unit description
InformUnitRoles(?Unit:ID)	Indicates which roles are the ones defined within a specific unit
InformRoleProfiles(?Role:ID)	Indicates all profiles associated to a specific role
InformRoleNorms(?Role:ID)	Provides all norms addressed to a specific role
Dynamic Services	
RegisterAgentRole(?Agent:ID, ?Role:ID, ?Unit:ID)	Creates a new <*entity, unit, role*> relationship. Private OMS service.
DeregisterAgentRole(?Agent:ID, ?Role:ID, ?Unit:ID)	Removes a specific <*entity, unit, role*> relation. Private OMS service.
AcquireRole(?Unit:ID, ?Role:ID)	Requests the adoption of a specific role within a unit
LeaveRole(?Unit:ID, ?Role:ID)	Requests to leave a role
Expulse(?Agent:ID, ?Unit:ID, ?Role:ID)	Forces an agent to leave a specific role

Table 2. OMS Services

The OMS offers a set of basic services for dynamical role adoption and the entry/exit of unit members, which are not directly accessible to agents, but are combined through compound services. The *basic services* for role adoption are *RegisterAgentRole* (that creates a new <*entity, unit, role*> relationship) and *DeregisterAgentRole* (that removes a specific <*entity, unit, role*> relationship). The OMS also offers a set of compound services that can be used by agents for adopting roles, leaving them and applying sanctions. These *compound services* are *AcquireRole*, *LeaveRole* and *Expulse*, detailed in Table 2. Publishing these services enables external agents to participate inside the system.

To sum up, the OMS is responsible for managing the life-cycle of the organizations. Thus, it includes services for defining structural components of organizations, i.e. roles, units and norms. These structural components could be dynamically modified over the lifetime of the organization. Moreover, it includes services for creating new organizations (i.e. creating new units), admitting new members within those organizations (i.e. acquiring roles) and member resigning (i.e. expulsing or leaving roles).

5 A simplified usage sample

In order to illustrate the usage of the THOMAS architecture, a simplified case-study for making flight and hotel arrangements is used (a more complete specification can be downloaded from the project's web-page[5]). This is a well known example that has been modeled by means of electronic institutions in previous works [10, 36]. The *Travel Agency* example is an application that facilitates the interconnection between clients (individuals, companies, travel agencies) and providers (hotel chains, airlines); delimiting services that each one can request or offer. The system controls which services must be provided by each agent. Internal functionality of these services is responsibility of provider agents. However, the system imposes some restrictions about service profiles, service requesting orders and service results.

The *Travel Agency* case-study has been modelled as a THOMAS organization composed of different agents that implement travel agency services. The *TravelAgency* unit is formed by two organizational units (*HotelUnit* and *FlightUnit*) which represent groups of agents, dedicated to hotels or flights, respectively. The *Customer* role requests system services and it is specialized into *HotelCustomer* and *FlightCustomer*. Similarly, the *Provider* role is in charge of performing services (hotel or flight search services) and it is also specialized into *HotelProvider* and *FlightProvider*. The *TravelAgency* organizational unit offers a *SearchTravel* service, which is internally specialized into *SearchHotel* and *SearchFlight*. The visibility of these internal services is limited to the members of the *TravelAgency* unit.

The scenario depicted in Figure 2 shows the set of service calls for registering new agents as service clients inside the *TravelAgency* organization. A new client agent C1, which has already been registered in the THOMAS platform, requests *SearchService* to SF for finding services of its interest (message 1). As a result, C1 obtains *SearchTravel* service identifier and a ranking value (message 2). Then, C1 employs *GetProfile* (message 3), which specifies that service clients must play *Customer* role inside the *TravelAgency* (message 4). Therefore, C1 must adopt the *Customer* role for demanding this service, requesting *AcquireRole* to OMS (messages 5 and 6).

Once C1 plays this customer role, it employs *GetProcess* service in order to know who are the service providers and how this service can be requested (message 7). However, there are none providers for the general *SearchTravel* service (message 8). Inside the *TravelAgency* unit, C1 requests *SearchService* again (message 9). In this case, SF returns *SearchFlight* and *SearchHotel* services because both services are accessible from the *TravelAgency* organization. Then C1 demands the profile of *SearchHotel* service (using *GetProfile*, message 11), since this service is more appropriated to its needs. Taking into account the *SearchHotel* profile (message 12), C1 requests adopting *HotelCustomer* role inside *HotelUnit*, demading the *AcquireRole* service to the OMS (messages 13 and14). The OMS checks all restrictions (ex. role compatibility, norms related

[5] http://www.dsic.upv.es/users/ia/sma/tools/Thomas

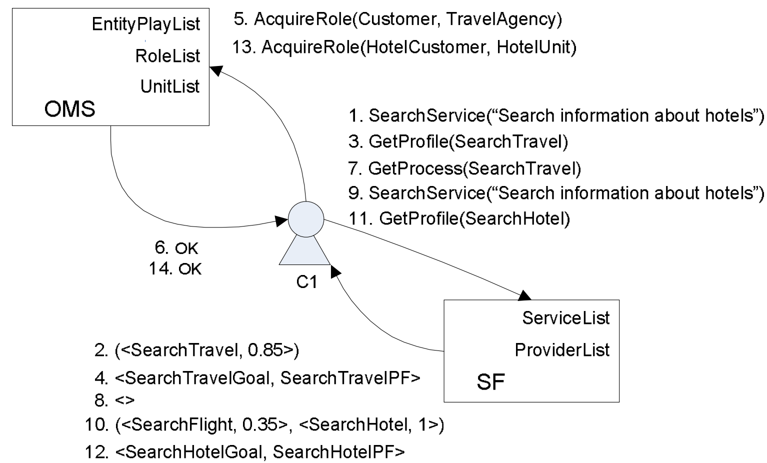


Fig. 2. A client registering scenario of a Travel Agency system

to the requested role) and registers the *<C1,HotelUnit, HotelCustomer>* relationship.

6 The THOMAS Framework: a prototype

Nowadays, a new agent platform based on the above described THOMAS abstract architecture is available. But, as this abstract architecture has been designed to work making use of any FIPA-compliant platform (as the Platform Kernel of the architecture) a new idea has arisen: the THOMAS Framework. This framework is composed by the OMS and SF modules of the abstract architecture, and its purpose is to try to obtain a product wholly independent of any internal agent platform, and as such, that is fully addressed for open systems. This framework is based upon the idea that no internal agent exists, and the architecture services are offered as web services. In this way, only the OMS and the SF are composing such framework (avoiding the use of the PK due to the lack of internal agents to control). Therefore, the THOMAS framework (see Figure 3) allows any agent to create a virtual organization with the structure and norms he wants, along with the demanding and offering services that he needs. The framework is in charge of the management of this organization structure, norms and life cycle, on one hand. On the other hand, it also controls the visibility of the offered and demanded services and the fulfillment of the conditions to use them. But, as it is fully addressed to open systems, the framework does not control the involved agents life-cycle, being all of them external to the framework.

The first version of this framework, v0.1, is available for download in the project's web-page. It implements the whole set of services described in the

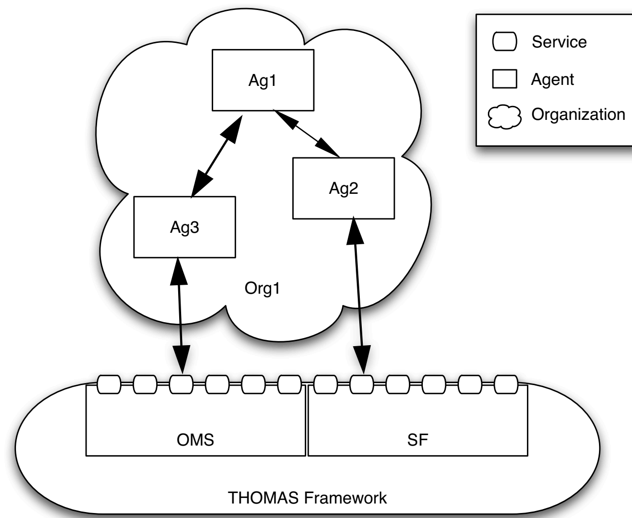


Fig. 3. THOMAS Framework

abstract architecture, with a basic support for norm management. This version has been used to check the feasibility of this approach with several examples using JADE and SPADE [15] agents.

7 Discussion and related works

In previous sections an abstract architecture for the development of real open multi-agent systems has been proposed. This proposal aims to instigate the total integration of two promising technologies, that is, multi-agent systems and service-oriented computing as the foundation of such virtual organizations. Both technologies try to deal with the same kind of environments formed by loose-coupled, flexible, persistent and distributed tasks [28]. Traditional web services, or even semantic web services, might be a valid solution when point-to-point integration of static-binded services is needed. But they are clearly not good enough for working in a changing environment, in which new services appear, have to be discovered and composed or adapted to different ontologies. The nature of agents, as intelligent and flexible entities with auto-organizative capabilities, facilitates automatic service discovery and composition. The vision of agents and organizations as service-provider entities is not new. The main effort in the integration of agents and web services is directed at masking services for redirection, aggregation, integration or administration purposes [22].

We can identify two approaches in previous related works: (i) direct integration of web services and agents by means of message exchange and (ii) the

consideration of agents as matchmakers for service discovering and composition. Works related with the former are the *Web Service Integration Gateway Service* (WSIG) architecture [23] and *AgentWeb Gateway* [35] which are based on the idea of using an intermediary entity between agents and web services. Another examples are *WSDL2JADE* [38] and *WS2JADE* [32] which provide agents with an interface for communicating directly with web services. Finally, *Web Service Agent Integration* [6] (WSAI) solves the problem in the opposite way by allowing web service clients to use agent services.

The latter approach is represented by works as [6], which presents an approach that complements the existing methods by considering the types of interactions and roles that services can be used in. Other approaches, such as [34], use the objective experience data of agents in order to evaluate their expectations from a service provider and make decisions using their own criteria and mental state. [37] presents a brokering protocol which consists of two complex reasoning tasks: discovery and mediation. Negotiation protocols are another mechanism normally used. In this case, participant agents negotiate about the properties of the services they request and provide to bind agreements and contracts with each other [9]. Our proposal goes beyond because agents can offer and invoke services in a transparent way from other agents, virtual organizations or entities, plus external entities can interact with agents through the use of the offered services.

Regarding organizational concepts in an open MAS system, we consider an agent organization as a social entity composed of a specific number of members which accomplish several distinct tasks or functions. These members are structured following some specific topology and communication interrelationship in order to achieve the main aim of the organization [3]. Agent organizations assume the existence of global goals, outside of the objectives of any individual agent, and they exist independently of agents [11].

Organizations have been usefully employed as a paradigm for developing agent systems [4, 19]. One of the advantages of the organization development is that systems are modeled with a high level of abstraction, so the conceptual gap between real world and models is reduced. Also, these kinds of systems provide the facilities to implement open systems and heterogeneous member participation [31].

Research into MAS organizations has ranged from basic organizational concepts, such as groups, communities, roles, functions [29, 39, 18, 33, 20]; organizational modeling [25, 13, 18, 14]; Human Organization Theory [21, 2]; structural topologies [24, 3]; to normative research, including internal representation of norms [30], deontic logics [12, 5] and institutional approaches [16]. Our work presents an approach which covers all the life-cycle management of an agent organization through the use of the OMS entity, which is in charge of the specification and administration of all the structural and dynamic components of an agent organization.

[6] http://www.agentcities.org/rec/00006/actf-rec-00006a.pdf

Finally, another key problem for open MAS development is the existence of real agent platforms that support organizational concepts. Over the last few years, many agent platforms and agent architectures have been proposed. A detailed comparison of these platforms, focusing on organizational concepts, can be found in [1]. Despite the large number of agent platforms in existence, the majority are lacking in the management of virtual organizations for dynamic, open and large-scale environments. Designers must implement nearly all of the organizational features by themselves, namely organization representation, control mechanisms, organization descriptions, AMS and DF extensions, communication layer, monitoring, organization modeling support and organizational API. These features are briefly explained as follows.

With respect to *organization representation*, a possible solution is that agents have an explicit representation of the organization which has been defined. This paper deals with this approach as S-Moise+ [26] and Ameli (EI platform) [16]. They have an explicit representation of the organization and both have similar architectures. Another feature well supported by *AMELI* and *S-Moise+* are *control mechanisms* that ensure the satisfaction of the organizational constraints. The organization should have an available *description* in a standard language. This allows external and internal agents to get specific information about the organization at run-time. This feature is not only useful in open systems, but also when considering a reorganization process. A good example of organization specification can be found in the *S-Moise+* platform.

One of the main lacks in current agent platforms is the *AMS and DF extension*. The AMS should have information on the existing organizations and their members. The DF should publish the services offered by agents individually and the services offered by organizations. Another important features are: the *communication layer*, as the kind of communication layer used in communicative acts is a very important feature; the *system monitoring*, i.e. the platform should offer a mechanism for monitoring the states of agents and organizations; the *modeling concepts support*, as the platform and the programming language should cover all of the concepts related to the virtual organization. For example, which types of topologies are defined within the platform, which kind of norms are modeled, etc. Not all platforms have a complete modeling concept support. For example *AMELI* is focused on the management of rules and norms but does not support the definition of complex topologies. *Jack Teams* platform allows the creation of composed "Teams" but it does not take into account other topologies.

Finally, the platform should offer an *organizational API* that makes it possible to create, destroy and modify organizations; consult and modify the organization description; add, query and delete agents of an organization; send messages to a whole organization, etc [3, 7]. Our proposal includes a platform fully addressed for open systems which has in mind all these factors trying to obtain a framework wholly independent of any internal agent platform.

8 Conclusions

The main contribution of this paper is the definition of an open architecture for large scale open multi-agent systems based on a service-oriented approach. As the previous mentioned discussion has shown, there exists a current research interest in the integration of agents and services, agents being complex entities that can handle the problem of service discovery and composition in dynamic and changing open environments. Moreover, current agent approaches are not organized into plain societies, but into structured organizations that enclose the real world with the society representation and ease the development of open and heterogeneous systems. Current agent architectures and platforms must integrate these concepts to allow designers to employ higher abstractions when modeling and implementing these complex systems. All of these concerns are gathered in the previously presented THOMAS proposal.

Moreover, the proposal has been implemented and is available for downloading in the project's web-page.

9 Acknowledgments

This work has been partially funded by TIN2008-04446, TIN2006-14630-C03-01, PROMETEO/2008/051, and GVPRE/2008/070 projects and CONSOLIDER-INGENIO 2010 under grant CSD2007-00022.

References

1. E. Argente, A. Giret, S. Valero, V. Julian, and V. Botti. Survey of MAS Methods and Platforms focusing on organizational concepts. In Vitria, J., Radeva, p. and Aguilo, I, editor, *Recent Advances in Artificial Intelligence Research and Development*, Frontiers in Artificial Intelligence and Applications, pages 309–316, 2004.
2. E. Argente, V. Julian, and V. Botti. Multi-agent system development based on organizations. *Electronic Notes in Theoretical Computer Science*, 150:55–71, 2006.
3. E. Argente, J. Palanca, G. Aranda, V. Julian, V. Botti, A. García-Fornes, and A. Espinosa. Supporting agent organizations. In *Proc. CEEMAS'07*, pages 236–245, 2007.
4. O. Boissier, J. Padget, V. Dignum, G. Lindemann, E. Matson, S. Ossowski, J. Sichman, and J. Vazquez-Salceda. *Coordination, Organizations, Institutions and Norms in Multi-Agent Systems*, volume 3913 of *LNCS (LNAI)*. Springer-Verlag, 2006.
5. J. Broersen, F. Dignum, V. Dignum, and J. Meyer. Designing a deontic logic for deadlines. In *Proc. 7th International Workshop on Deontic Logic in Computer Science*, 2004.
6. C. Caceres, A. Fernandez, S. Ossowski, and M. Vasirani. Role-based service description and discovery. In *International Joint Conference on Autonomous Agents and Multi-Agent Systems*, 2006.
7. N. Criado, E. Argente, V. Julian, and V. Botti. Organizational services for spade agent platform. In *Proc. 6th International Workshop on Practical Applications on Agents and Multi-Agent Systems (IWPAAMS07)*, 2007.

8. M. P. D. Martin and M. Wagner. Towards semantic annotations of web services: Owl-s from the sawsdl perspective. 2007.
9. J. Dang and M. Hungs. Concurrent multiple-issue negotiation for internet-based services. In *IEEE Internet Computing*, number Vol.10 - 6, pages 42–49, 2006.
10. F. Dignum, V. Dignum, J. Thangarajah, L. Padgham, and M. Winikoff. Open Agent Systems?? In *8th International Workshop on Agent Oriented Software Engineering (AOSE-07)*, 2007.
11. V. Dignum and F. Dignum. A landscape of agent systems for the real world. Technical report 44-cs-2006-061, Institute of Information and Computing Sciences, Utrecht University, 2006.
12. V. Dignum and F. Dignum. A logic for agent organization. In *Proc. FAMAS@Agents'007*, 2007.
13. V. Dignum, J. Meyer, H. Weigand, and F. Dignum. An organization-oriented model for agent societies. In M. P. . B. Y. E. Lindemann, D. Moldt, editor, *International Workshop on Regulated Agent-Based Social Systems: Theory and Applications (RASTA'02)*, pages 31–50, 2002.
14. V. Dignum, J. Vazquez-Salceda, and F. Dignum. Omni: Introducing social structure, norms and ontologies into agent organizations. *LNAI 3346*, 2005.
15. M. Escrivà, J. Palanca, G. Aranda, A. García-Fornes, V. Julian, and V. Botti. A jabber-based multi-agent system platform. In *Proc. of AAMAS06*, pages 1282–1284, 2006.
16. M. Esteva, J. Rodriguez-Aguilar, C. Sierra, J. Arcos, and P. Garcia. *On the Formal Specification of Electronic Institutions*, pages 126–147. Lecture Notes in Artificial Intelligence 1991. Springer-Verlag, 2001.
17. D. M. et al. Owl-s: Semantic markup for web services. `http://www.w3.org/Submission/2004/OWL-S`, 2004.
18. J. Ferber and O. Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS'98)*, pages 128–135. IEEE Computer Society, 1998.
19. J. Ferber, O. Gutknecht, and F. Michel. From Agents to Organizations: an Organizational View of Multi-Agent Systems. In P. Giorgini, J. Muller, and J. Odell, editors, *Agent-Oriented Software Engineering VI*, volume LNCS 2935 of *Lecture Notes in Computer Science*, pages 214–230. Springer-Verlag, 2004.
20. L. Gasser. *Perspective on Organizations in Multi-Agent Systems*, pages 1–16. Springer-Verlag, 2001.
21. P. Giorgini, M. Kolp, and J. Mylopoulos. Multi-agent architectures as organizational structures. *Autonomous Agents and Multi-Agent Systems*, 13(1):3–25, 2006.
22. D. Greenwood and M. Calisti. Engineering web service - agent integration. In *IEEE International Conference on Systems, Man and Cybernetics*, number Vol. 2, pages 1918– 1925, 2004.
23. D. Greenwood, M. Lyell, A. Mallya, and H. Suguri. The ieee fipa approach to integrating software agents and web services. In *AAMAS '07: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–7, New York, NY, USA, 2007. ACM.
24. B. Horling and V. Lesser. A survey of multiagent organizational paradigms. *The Knowledge Engineering Review*, 19:281–316, 2004.
25. B. Horling and V. Lesser. Using ODML to Model Multi-Agent Organizations. In *IAT05: Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, 2005.

26. J. Hubner, J. Sichman, and O. Boissier. S-Moise+: A middleware for developing organised multi-agent systems. In *Proc. Int. Workshop on Organizations in Multi-Agent Systems, from Organizations to Organization Oriented Programming in MAS*, volume 3913 of *LNCS*, pages 64–78, 2006.
27. M. Huhns and M. Singh. Reseach directions for service-oriented multiagent systems. *IEEE Internet Computing*, Service-Oriented Computing Track. 9(1), 2005.
28. M. Huhns and M. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, Service-Oriented Computing Track. 9(1), 2005.
29. N. R. Jennings and M. Wooldridge. Agent-oriented software engineering. *Handbook of Agent Technology*, 2002.
30. F. Lopez, M. Luck, and M. d'Inverno. A normative framework for agent-based systems. *Computational and Mathematical Organization Theory*, 12:227–250, 2006.
31. X. Mao and E. Yu. Organizational and social concepts in agent oriented software engineering. In *AOSE IV*, volume 3382 of *Lecture Notes in Artificial Intelligence*, pages 184–202, 2005.
32. T. Nguyen and R. Kowalczyk. Ws2jade: Integrating web service with jade agents. Technical Report SUTICT-TR2005.03, Centre for Intelligent Agents and Multi-Agent Systems, Swinburne University of Technology, 2005.
33. J. Odell, M. Nodine, and R. Levy. A Metamodel for Agents, Roles, and Groups. In J. M. James Odell, P. Giorgini, editor, *Agent-Oriented Software Engineering (AOSE) V*, Lecture Notes in Computer Science. Springer, 2005.
34. M. Sensoy, C. Pembe, H. Zirtiloglu, P. Yolum, and A. Bener. Experience-based service provider selection in agent-mediated e-comerce. In *Engineering Applications of Artificial Intelligence*, number 3, pages 325–335, 2007.
35. M. O. Shafiq, A. Ali, H. F. Ahmad, and H. Suguri. Agentweb gateway - a middleware for dynamic integration of multi agent system and web services framework. In *14th IEEE International Workshops on Enabling Technologies (WETICE 2005), 13-15 June 2005, Linköping, Sweden*, pages 267–270. IEEE Computer Society, 2005.
36. C. Sierra, J. Thangarajah, L. Padgham, and M. Winikoff. Designing Institutional Multi-Agent System. In *7th International Workshop on Agent Oriented Software Engineering (AOSE'06), LNCS 4405*, number 1-2, pages 84–103, 2006.
37. K. Sycara, M. Paolucci, J. Soudry, and N. Srinivasan. Dynamic discovery and coordination of agent-based semantic web services. In *IEEE Internet Computing*, number Vol.8 - 3, pages 66–73, 2004.
38. L. Z. Varga and Á. Hajnal. Engineering web service invocations from agent systems. In *Multi-Agent Systems and Applications III, 3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003, Prague, Czech Republic, June 16-18, 2003, Proceedings*, volume 2691 of *Lecture Notes in Computer Science*, pages 626–635. Springer, 2003.
39. F. Zambonelli and H. Parunak. From design to intention: Signs of a revolution. In *Proc. 1st Int. Joint Conference on Autonomous Agents and MultiAgent Systems*, pages 455–456, 2002.

Multi-agent systems: Modeling and Verification using Hybrid Automata

Ammar Mohammed and Ulrich Furbach

Universität Koblenz-Landau, Computer Science Department, Koblenz, Germany,
{ammar,uli}@uni-koblenz.de

Abstract. Hybrid automata are used as standard means for the specification and analysis of dynamical systems. Several researches have approached them to formally specify reactive Multi-agent systems situated in a physical environment, where the agents react continuously to their environment. The specified systems, in turn, are formally checked with the help of existing hybrid automata verification tools. However, when dealing with multi-agent systems, two problems may be raised. The first problem is a state space problem raised due to the composition process, where the agents have to be parallel composed into an agent capturing all possible behaviors of the multi-agent system prior to the verification phase. The second problem concerns the expressiveness of verification tools when modeling and verifying certain behaviors. Therefore, this paper tackles these problems by showing how multi-agent systems, specified as hybrid automata, can be modeled and verified using constraint logic programming(CLP). In particular, a CLP implementation is presented to show how the composition of multi-agent behaviors can be captured dynamically during the verification phase. This can relieve the state space complexity that may occur as a result of the composition process. Additionally, the expressiveness of the CLP implementation model flexibly allows to not only model multi-agent systems, but also to check various properties by means of the reachability analysis. Experiments are promising to show the feasibility of our approach.

1 Motivation

Specifying behavior for (physical) multi-agent systems is a sophisticated and demanding task, because of the high complexity of the interactions among agents and the dynamics of the environment. An important aspect of multi-agent systems is that the agents interact with a physical environment. Such interactions typically consist of continuous changes of behaviors of agents (e.g. a movement of a robot, or an agent is waiting for occurrence of an event), as well as discrete changes of behaviors. Those scenarios can be captured by the use of hybrid automata [12]. Here the discrete changes are modeled using a form of transition diagrams dialect like statecharts [24], while the continuous changes are modeled using differential equations. Hybrid automata formal semantics make them accessible to formal validation of systems in safety critical environments. Thus, it is possible to prove desirable features as well as the absence of unwanted properties for the modeled systems automatically with the help of hybrid automata verification tools [13, 8, 3].

Hybrid automata can be used to model multi-agent systems that are defined through their capability to continuously react to a physical environment, while respecting some time constraints. Therefore, several researches for example [22, 6, 7, 9], have approached hybrid automata as a framework to model reactively multi-agent plans, where the time is critical. There are authors, for example [18], who have approached multi-agent systems with a simple form of hybrid automata that are called timed automata [2]. Nevertheless, two problems occur when applying hybrid automata to multi-agent systems. Firstly, multi-agent systems are specified as a network of synchronized hybrid automata that have to be parallel composed statically into an automaton (synonymy agent). By statically we mean that agents have to be parallel composed prior to the verification phase. Technically, the composition of hybrid automata is obtained from the Cartesian product of the number of states of all concurrent automata, unless the automata have mutual synchronization messages. In this case, the states have to be considered simultaneously. As a result of the composition process, an agent captures all possible behaviors that may occur in the multi-agent systems. In turn, the resulting composed agent afterwards is checked by hybrid automata verification tools. Consequently, this composition process may lead to a state explosion problem.

The second problem concerns the expressiveness of the modeling tools. Standard hybrid automata tools are not flexible enough to model multi-agent systems, because they are special purpose tools, which model the agents' decision depending on the evaluation of continuous dynamics. However,there are favorable situations of modeling multi-agent systems where the agents' decision steps do not depend on the evaluation of continuous dynamics, but on evaluation functions (e.g. shortest distance, max, or min) happening during the continuous dynamic. Imagine, for example, an agent who wants to cooperate with the nearest agent to conduct certain tasks in a rescue team of a multi-agent system. To our knowledge, this type of decision making is beyond the capabilities of the current hybrid automata verification tools. Therefore it is necessary to have expressive tools that can handle such situations. Ideally, modeling tools are favorable when they are flexibly able to verify the systems' requirements.

To this end, the purpose of this paper is to cope with the mentioned problems when approaching hybrid automata to model multi-agent systems. In particular, we present a novel approach which models hybrid automata based on constraint logic programming, which is appropriate to represent multi-agent systems specified as hybrid automata. The novelty of the presented approach is the composition of hybrid automata that is built on the fly, where only the reached behaviors are captured dynamically, instead of building all possible behaviors in advance. On the other hand, the expressiveness of CLP does not only allow us to model multi-agent systems, but also to check various properties by representing requirements with a suitable query. We show the feasibility of our approach with experimentations on standard benchmarks taken from the hybrid automata context.

1.1 Overview on the Rest of the Paper

In summary, the main contributions of this paper are as follows: First, a lean but effective implementation of hybrid automata, suitable to multi-agent systems, is presented. Second, compositions of automata do not have to be computed explicitly, which avoids the state explosion problem. Last but not least, by employing CLP, constraints can be

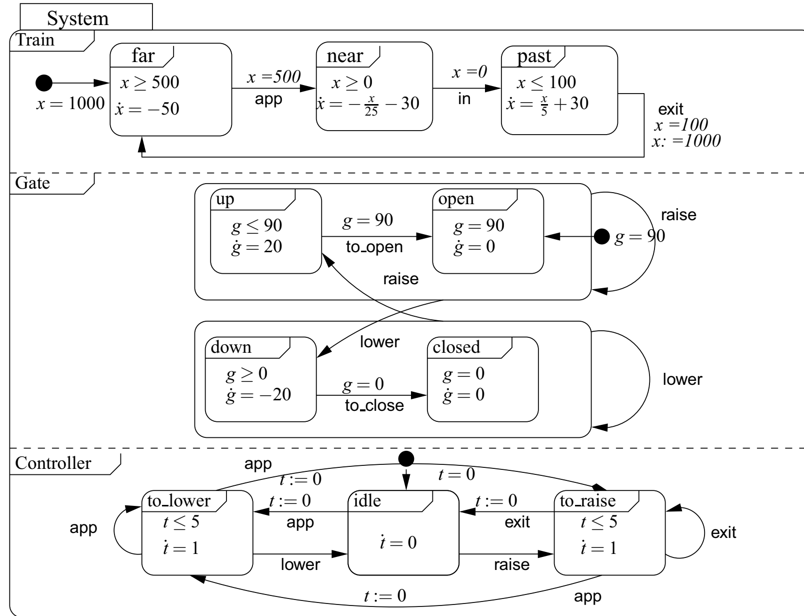


Fig. 1. Specification of the train gate controller as hybrid automata.

derived automatically, under which certain states of the system can be tested for reachability. This enhances standard model checking methodologies.

In the sequel, we first introduce a running example that will be used throughout the paper to illustrate our approach in Sec. 2. Then hybrid automata syntax and semantics are discussed in Sec. 3. In Sec. 4 a CLP implementation model is discussed, before showing how to specify and verify requirements in Sec. 5. The evaluation of our CLP implementation model is discussed in Sec. 6. Then Sec. 6.1 briefly reviews related works, before we end up with the conclusion Sec. 7

2 Running Example

Before we present both syntax and semantic of hybrid automata, we first introduce an illustrating running example that we use throughout the paper, before we shows the basics formalism which we use to demonstrate the CLP implementation.

A train gate controller [14] is a reactive multi-agent system consisting of three agent components: the train, the gate, and the controller. In this system, a road is crossing a train track, which is guarded by a gate, which must be lowered to stop the traffic when the train approaches, and raised after a train passed the road. The gate is supervised by a controller that has the task to receive signals from the train and to issue lower or raise signals to the gate. Initially, a train is at a distance of 1000 meters away from the gate and moves at a speed 50 meter per second. At 500 meters, a sensor on the tracks detects the train, sending a signal *app* to the controller. The train slows down, obeying the

differential equation $\dot{x} = -\frac{x}{25} - 30$. After a delay of five seconds, which is modeled by the variable *t*, the controller sends the signal *lower* to the gate, which begins to descend from 90 degrees to 0 degrees at a rate of -20 degrees per second. After crossing the gate, the train accelerates according to the differential equation $\dot{x} = \frac{x}{5} + 30$. A second sensor placed 100 meters past the crossing detects the leaving train, sending a signal *exit* to the controller. After five seconds, the controller raises the gate.

The specification of the previous multi-agent system is graphically illustrated as concurrent hybrid automata in Fig. 1. The variable x represents the distance of the train from the gate. The variable t represents the delay time of the controller, while the position of the gate in radius degrees is represented by the variable g.

3 Hybrid Automata Preliminaries

In this section, we show the basics syntax and the semantics of hybrid automata.

3.1 Hybrid Automaton: Syntax

A hybrid automaton is represented graphically as a state transition diagram dialect like statecharts, augmented with mathematical formalisms on both transitions and locations. Formally speaking, a hybrid automaton (agent in continuous domain) is defined as follows.

Definition 1 (basic components). *A hybrid automaton is a tuple* $H = (X, Q, Inv, Flow, E, Jump, Reset, Event, Init)$ *where:*

- $X \subseteq \Re^n$ *is a finite set of n real-valued variables that model the continuous dynamics.*
- Q *is a finite set of control locations. For example, the train automaton (Fig. 1) has the locations far, near,and past.*
- $Inv(q)$ *is the invariant predicate, which assigns a constraint on variables X for each control location* $q \in Q$*. The control of a hybrid automaton remains at a location* $q \in Q$*, as long as* $Inv(q)$ *holds. For instance, the location* far *in the train automaton has the invariant* $x \geq 500$
- $Flow(q)$ *is the flow predicate on variables X for each control location* $q \in Q$*, which defines how the the variables in X evolve over the time at location q. It constrains the time derivative of the continuous part of the variables at location q. In the graphical representation, a flow of a variable x is denoted as* $\dot{x}$*. For example,* $\dot{x} = \frac{x}{5} + 30$ *describes the speed of the train at the location* past *in the train automaton (Fig. 1). If* $\dot{x} = c$*, then the hybrid automaton is called linear (a special case of linear hybrid automata are a timed automata [2], where* $c = 1$*). if* $\dot{x} = c_1 x + c_2$*, then a hybrid automaton is called non-linear.*
- $E \subseteq Q \times Q$ *is the discrete transition relation over the control locations. Each edge* $e \in E$ *is augmented by the following annotations:*

  Jump: *jump condition (guard), which is a constraint over X that must hold to fire transitions. Upon firing a transition e, values of the variables X may be changed by executing a specific action.*

Reset: *is a constraint, which may reset the variables by executing a specific assignments. For example, the variable X in the train automaton on the transition between locations* past *and* far *is reset to* $X := 1000$. *Resetting variables are omitted on transition, if the values of the variables do not change before the control goes from a location to another location.*

Event: *synchronization label, used to synchronize concurrent automata. For instance, the train automaton contains the synchronization labels* app, in, *and* exist*, which must be synchronized with all automata sharing the same synchronization labels. These synchronization labels define the composition of the automata.*

– *Init is the initial condition that assigns an initial values to the variables X to each control location $q \in Q$. For example,* $x = 1000$ *is the initial condition of the train automaton.*

3.2 Hybrid Automaton: Semantics

Informally speaking, the semantics of a hybrid automaton is defined in terms of a labeled transition system between states, where a state consists of the current location of the automaton and the current valuation of the real variables. To formalize the semantics of the hybrid automaton, we first need to define the concept of a hybrid automaton's state.

Definition 2 (State). *At any instant of time, a state of a hybrid automaton is given by $\sigma_i = \langle q_i, v_i, t \rangle$, where $q_i \in Q$ is a control location, v_i is the valuation of the real variables, and t is the current time. A state $\sigma_i = \langle q_i, v_i, t \rangle$ is admissible if $Inv(q_i)[v_i]$ holds.*

A state transition system of a hybrid automaton H starts with the *initial state* $\sigma_0 = \langle q_0, v_0, 0 \rangle$, where the q_0 and v_0 are the initial location and valuations of the variables respectively. For example, the initial state of the *train* (see Fig. 1) can be specified as $\langle far, 1000, 0 \rangle$.

Intuitively, an execution of a hybrid automaton corresponds to a sequence of transitions from a state to another. In fact, a hybrid automaton evolves depending on two kinds of transitions: continuous transitions, capturing the continuous evolution of states, and discrete transitions, capturing the changes of location. More formally, we can define hybrid automaton semantics as follows.

Definition 3 (Operational Semantic). *A transition rule between two admissible states $\sigma_1 = \langle q_1, v_1, t_1 \rangle$ and $\sigma_2 = \langle q_2, v_2, t_2 \rangle$ is defined as follows:*

discretely: *iff $t_1 = t_2$ and $Jump(v_1)$ holds, then variables are reset at location q_2 such that, $Inv(q_2)[v_2]$ holds. In this case an event $a \in Event$ may occur.*

continuously(time delay): *iff $q_1 = q_2$, and ($t_2 - t_1 > 0$) is the duration of time passed at location q_1, during which the invariant predicate $Inv(q1)$ continuously holds, v_1, v_2 are the variable valuations according to the flow predicate $Flow(q1)$.*

In principle, an execution of a hybrid automaton corresponds to a sequence of transitions from one state to another, therefore we define the valid run as follows.

Definition 4 (Run). *A run of hybrid automaton $\sum = \sigma_0\sigma_1\sigma_2,..,$ is a finite or infinite sequence of admissible states, where σ_0 is the initial state.*

In a run $\sum$, the transition from a state σ_i to a state σ_{i+1} is related by either discrete or continuous transition, according to Def. 3.

It should be noted that the continuous change in the run may generate an infinite number of reachable states. It follows that state-space exploration techniques require a symbolic representation system for the sets of states that have to be manipulated (in this paper, CLP represents the infinite states symbolically as a finite interval). we call the symbolic interval as region. Consequently, the set of all reachable states at location $q \in Q$ can be represented as $\langle q, V, Time \rangle$, where V and $Time$ represent the reachable region and time at location q respectively. Now, the run of hybrid automata can be re-stated as a form of reachable regions, where the change from one region to another one is fired using a discrete step.

The operational semantics is the basis for verification of hybrid automata. In particular, model checking of a hybrid automaton is defined in terms of reachability analysis of the hybrid automaton.

Definition 5 (Reachability). *A state σ_j is reachable from a state σ_i, if there is a sequence of admissible states starting from σ_i and ending in σ_j. A state σ_j is called reachable if it can be reached from the initial state σ_0.*

The classical method to compute the reachable states consists of performing a state space exploration of the system, starting from a set containing only the initial state and spreading the reachability information along control locations and transitions until a stable region is obtained. Stabilization is detected by testing if the current region is included in the union of the reached regions obtained in previous steps. It is worth mentioning that checking reachability for hybrid automata is generally undecidable even for a simple class of hybrid automaton [15].

3.3 Hybrid Automata: Composition

To model concurrent systems, hybrid automata can be extended by parallel composition. Basically, parallel composition of hybrid automata can be used for specifying larger systems (multi-agent systems), where a hybrid automaton is given for each part of the system, and communication between the different parts may occur via shared variables and synchronization labels. Technically, the parallel composition of hybrid automata is obtained from the different parts using a product construction of the participating automata. The transitions from the different automata are interleaved, unless they share the same synchronization label. In this case, they are synchronized during the execution. As a result of the parallel composition, an automaton is created, which captures the behavior of the entire system.

Intuitively, if A and B are two automata, then a run of the composed hybrid automata is a sequence $\sum = \sigma_0\sigma_1\sigma_2,..$ and $\sigma_i = \langle (q,r),(V,U),T \rangle$, where q and r are the control locations of both automata A and B respectively, whereas V and U are the respective reached regions during the time interval T in both locations q and r. The previous means that a state of composed automata consists a vector of the current locations of the hybrid

automata - one for each automaton - along with the valuations of the respective variables and time.

4 CLP Model

In the following, we will show how to encode the hybrid automata described in the previous section as a Constraint Logic Program *CLP* [19]. There are diverse motivations for choosing CLP. Firstly, hybrid automata can be described as a constraint system, where the constraints represent the possible flow, invariant, and transitions. Further, constraints can be used to characterize a certain part of the state space (e.g, the set of initial state or a set of unsafe state). Secondly, there are close similarities in operation semantics between CLP and hybrid system. Ideally, state transition systems can be represented as a logic program, where the set of reachable states can be computed. Moreover, constraints enable us to represent infinite states symbolically as a finite interval. Hence, the constraint solver can be used to reason about the reachability of a particular state. In addition, CLP is enriched with many efficient constraint solvers for interval constraints and symbolic domains, where the interval constraints can used to represent the continuous evolution, whereas symbolic domains are appropriate to represent the synchronization events (communication messages).

Our implementation prototype was built using ECLiPSe Prolog [21]. A preliminary implementation model was introduced in [23]. The prototype follows both the formal syntax and semantics definition of hybrid automata (Def. 1) and the semantics of the labeled transition semantics of hybrid automata. We start modeling each hybrid automaton individually. Therefore, we start by modeling locations that are implemented in the `automaton` predicate, ranging over the respective locations of the automaton, real-valued variables, and the time:

```
automaton(+Location,?Vars,+Vars0,+T0,?Time):-
                Vars#c2(Vars0,T0,Time),
                c1(Inv),Time $>=T0.
```

Here, `automaton` is the name of automaton itself, and `Location` represents the current locations of the automaton. `Vars` is a list of real variables participating in the automata, whereas `Vars0` is a list of the correspondent initial values. `c1(Invs)` is the invariant constraint inside the location, and the constraint predicate `Vars # c2(Vars0,T0,Time)`, where $\# \in \{<, \leq, >, \geq, =\}$ are constraints, which represent the continuous flows of the variables in `Vars` wrt. time `T0` and `Time`, given initial values `Vars0` of the variables `Vars` at the start of the flow. `T0` is the initial time at the start of continuous flow, while `(Time-T0)` represents the delay inside the location. The following is an example showing the implementation of location *far* in automaton *train* Fig. 1.

```
train(far,[X],[X0],T0,Time):-
     X $= X0-50*(Time-T0),
     X $>=500, Time $>=T0.
```

According to operational semantics defined in Def. 3, a hybrid automaton has two kinds of transitions: *continuous* transitions, capturing the continuous evolution of variables, and *discrete* transitions, capturing the changes of location. For this purpose, we encode transition systems into the CLP *evolve* predicate as follows.

```
evolve(Automaton,(L1,Var1),(L2,Var2),T0,Time,Event) :-
     continuous(Automaton,(L1,Var1),(L1,Var2),T0,Time,Event);
     discrete(Automaton,(L1,Var1),(L2,Var2),T0,Time,Event).
```

When a discrete transition occurs, it gives rise to update the initial variables from `Var1` into `Var2`, where `Var1` and `Var2` are the initial variables of locations `L1` and `L2` respectively. Otherwise, a delay transition is taken using the predicate *continuous*. It is worth noting that there are infinite states due to the continuous progress. However, this can be handled efficiently as interval constraint that bounds the set of infinite reachable state as a finite interval (i.e., $0 \leq X \leq 250$).

Additionally, an `event` $\in Event_{Automaton}$ is associated with each transition, a variable *Event*, which is used to define the parallel composition from the automata individual sharing the same event. `Event` ranges over symbolic domains. It guarantees that whenever an automaton generates an event, the corresponding synchronized automata have to be taken into consideration simultaneously. When an automaton generates an event, the symbolic domain solver will exclude all the domain values that are not coincident with the generated event from the automata having the common event. This means that only one event is generated at a time. Consequently, it shows that the automata composition can be implicitly constructed efficiently on the fly, during the computation. Appropriately, the way that we construct the composition helps us to construct complex automata in terms of simpler ones. The following is the general implementation of the `discrete` predicate, which defines transitions between locations.

```
discrete(automaton,(+Loc1,+Var1),(?Loc2,?Var2),T0,Time,-Event):-
        automaton,(Location,Var1,Var,T0,Time),
        jump(Var) , reset(Var2)
        Event &::events,Event &=event.
```

Here, `event` must be a member in $Event_{Automaton}$. The & symbol is the constraint relation for symbolic domains (library *sd* in ECLiPSe Prolog), while the $ symbol (see below) marks interval constraints (library *ic*).

The following is an instance showing the implementation of the `discrete` predicate between locations *far* and *near* in automaton *train*.

```
discrete(train,(far,[X0]),(near,[XX0]),T0,Time,Event):-
          train(far,[X0],[X],T0,Time),
          X $=500, XX0 $=X,
          Event &::events,Event &=app.
```

The description of the above *discrete* predicate means that a transition between the locations *far* and *near* in the *train* automata takes place if the continuous variable `X` , based on the initial value `X0`, satisfies the jump condition given as `X=500`. If such a case occurs, then the new variable, denoted `XX0`, is updated, and the event *app* is fired. The executed events afterwards synchronize the *train* automaton with the automata sharing the same event. The & symbol is the constraint relation for symbolic domains (library *sd*

in ECLiPSe Prolog), while the $ symbol (see below) marks interval constraints (library *ic*). Once the transition rules have been modeled, a driver program needs to be supplied:

```
driver((+L1,+Var01),(+L2,+Var02),...,(+Ln,+Var0n),+T0,
[(L1,L2,..,Ln,-Var1,-Var2,..,-Varn,-Time,-Event)|-NextRegion]) :-
     automaton1(L1,Var1,Var01,T0,Time1),
     automaton2(L2,Var2,Var02,T0,Time2),
     ... ,
     automatonn(Ln,Varn,Var0n,T0,Timen),
     Time1 $=Time2, Time1 $=Time3, ..., Time1 $=Timen,
     evolve(automaton1,(L1,Var01),(NextL1,Nvar01),T0,Time1,Event),
     evolve(automaton2,(L2,Var02),(NextL2,Nvar02),T0,Time1,Event),
     ... ,
     evolve(automatonn,(Ln,Var0n),(NextLn,Nvar0n),T0,Time1,Event),

driver((NextL1,Nvar01),(NextL2,Nvar02),...,(NextLn,Nvar0n),Time1,NextRegion).
```

The `driver` is a simulator predicate that is responsible to generate and control the execution behavior of the concurrent hybrid automata, as well as to provide the reachable states symbolically.

Recall again, `Event` is a symbolic domain variable shared among all automata, where it is used by the solver to ensure that only one event is generated at a time. All automata sharing the same events have to be synchronized. During the computational process, each automaton A_i ,$1 \leq i \leq n$, produces a time $Time_i$, which is needed to jump from the automaton's current location into another location. Constraining these times of each automaton together leads to a time holding the minimum time among them. This minimum time,manipulated by the constraints solver, is least time needed to fire an event. Consequently, the predicate `evolve`, based on this time,alternates each automaton between continuous or discrete transition.

The last argument of the predicate `driver` is the list of finite reached regions. At each step of the driver, a region, of the form $\langle locations, Variables, Time \rangle$ symbolically represents the set of reached states and time to each control location as mathematical constrains. Additionally, each region contains the event generated at the end of reached regions before the control goes to another region using a discrete step.

The driver has to be invoked with a query starting from the initial states of the hybrid automata. An example showing how to query the driver on our multi-agent scenario (Fig. 1) takes the form:

```
?- driver((far,1000),(open,90),(idle,0),0,Reached).
```

Reachable states should contain only those variables, which are important for the verification of a given property. Therefore, the last argument list of the predicate `driver` can be expanded or shrunk as needed to contain the significant variables.

5 Verification as Reachability Analysis

Now we have an executable constraint based specification, which can be used to verify properties of our multi-agent team. Several properties can now be investigated. In

particular, one can check properties on states using reachability analysis of hybrid automata. Fundamentally, the reachability analysis consists of two basic steps: computing the state space of the automaton under consideration (in our case, this is done using the predicate `driver`), and searching for states that satisfy or contradict given properties. In terms of *CLP*, a state is reached iff the constraint solver succeeds in finding a satisfiable solution for the constraints representing the intended state. In other words, assuming that *Reached* represents the set of all reachable states computed by the *CLP* model from an initial state, then the reachability analysis can be generally specified, using *CLP*, by checking whether `Reached` $\models \Psi$ holds, where Ψ is the constraint predicate that describes a property of interest. In practice, many problems to be analyzed can be formulated as a reachability problem. For example, a safety requirement can be checked as a reachability problem, where Ψ is the constraint predicate that describes forbidden states, and then the satisfiability of Ψ wrt. `Reached` is checked. For instance, one can check that the state, where the train is near at distance `X=0` and the gate is open, is a disallowed state. Even a stronger condition can be investigated, namely that the state where the train is near at distance `X=0` and the gate is down, is a forbidden state. The *CLP* computational model, with the help of the standard Prolog predicate *member*/2, gives us the answer *no* as expected, after executing the following query:

```
?- driver((far,1000),(open,0),(idle,0),0,Reached),
   member((near,down,_,Time,_,X,),Reached), X $= 0.
```

Other properties concerning the reachability of certain states can be verified similarly.

Fundamentally, different properties can be checked in this framework As previously demonstrated, the set of reachable states `Reached` contains the set of finite, reachable regions. Within each region, the set of all states is represented symbolically as a mathematical constraint, together with the time delay. Therefore, constraint solvers ideally can be used to reason about the reachability of interesting properties within some region. For example, an interesting property is to find the shortest distance of the train to the gate before the gate is entirely closed. This can be checked by posing the following query:

```
?- driver((far,1000),(open,0),(idle,0),0,Reached),
   member((near,_,_,Time,to_close,_),Reached), get_max(Time,Tm),
   member((near,_,_,Tm,_,X),Reached), get_min(X,Min).
```

The previous query returns `Min=104.8` meters, which is the minimum distance of the train that the model guarantees before the gate is completely closed.

Since the events and time are recorded particularly at reached regions, verifying timing properties or computing the delay between events are further tasks that can be done within the reachability framework too. For instance, we can find the maximal time delay between *in* and *exit* events, by stating the following query:

```
?- driver((far,1000),(open,0),(idle,0),Reached),
      append(A,[(past,_,_,Time1,exit,_)|_],Reached),
      append(B,[(near,_,_,Time2,in,_)|_],A),
      get_max(Time1,Tmax1),get_max(Time2,Tmax2),
      Delay $= Tmax1-Tmax2.
```

where the predicate *append*/3 is the standard Prolog predicate. The constraint solver answers `yes` and yields `Delay=2.554`. This value means that the train needs at most `2.554` seconds to be in the critical crossing section before leaving it. Similarly, other timing properties can be verified.

6 Experimental Results

In the previous section, we have demonstrated how different properties can be verified within the *CLP* implementation framework.

We did several experiments comparing our approach with HyTech [16]. We chose HyTech as a reference tool, because it is one of the most well-known tools for the verification of hybrid automata, and it tackles verification based on reachability analysis similar to the approach in this paper. In HyTech however, the automata working in parallel are composed before they are involved in the verification phase. Obviously, this may lead to state explosion as stated earlier.

Now to use our approach to model and verify multi-agent systems, specified as hybrid automata, we have to demonstrate the feasibility of our proposed approach by experiments taken from the hybrid automata context. Therefore, we will refer to standard benchmarks of verification of real-time systems. Querying these benchmarks to check safety properties (cf. Fig. 2). First, in the *scheduler* example [11], it is checked whether a certain task (with number 2) never waits. Second, in the *temperature control* example [1], it has to be guaranteed, that the temperature always lies in a given range. Third, in the *train gate controller* example [13], it has to be ensured that the gate is closed whenever the train is within a distance less than 10 meter toward the gate. In the *water level* example [1, 11] the safety property is to ensure that the water level is always between given thresholds (1 and 12). Last but not least, a non-linear version of both train gate controller (described throughout this paper) and of the thermostat are taken from [14]. The safety property of the former one is the same as in the linear version, whereas in the second one we need to prove that the temperature always lies between 0.28 and 3.76. For more details on the examples, the reader is referred to the cited literature.

Example	HyTech	CLP
Scheduler	0.12	0.07
Temperature Controller	0.04	0.02
Train Gate Controller	0.05	0.02
Water Level	0.03	0.01
Train Gate Controller2	-	0.02
Thermostat	-	0.01

Fig. 2. Experimental results.

The symbol $-$ in Fig. 2 indicates that the example is inadequate to HyTech. This is because HyTech can not treat a non-linear dynamic directly.

When comparing HyTech to the approach depicted in this paper, several issues have to be taken into consideration. The first issue concerns the expressiveness of the dynamical model. HyTech restricts the dynamical model to linear hybrid automata in which the continuous dynamics is governed by differential equations. The nonlinear dynamics e.g. of the form $\dot{x} \bowtie c1 * x + c2$, where $c1, c2 \in \Re, c1 \neq 0, \bowtie \in \{<, \leq, >, \geq, =\}$ are first approximated either by a linear phase portrait or clock translation [17]. Then, the verifica-

tion phase is done on the approximated model. On the other hand, *CLP* is more expressive, because it allows more general dynamics. In particular, *CLP* can directly handle dynamics expressible as a combination of polynomials, exponentials, and logarithmic functions explicitly without approximating the model. For instance the last equation can be represented in *CLP* form as $X \ \$\!\bowtie X0 - c2/c1 + c2/c1 * exp(c1 * (T - T0))$, where $(T - T0)$ is the computational delay. Although clearly completeness cannot be guaranteed, from a practical point of view, this procedure allows to express problems in a natural manner. The *CLP* technology can be fully exploited; it suspends such complex goals until they become solvable.

Another issue that should be taken into account is the type of verifiable properties. HyTech cannot verify simple properties that depend on the occurrence of events, despite of the fact that synchronization events are used in the model. On the other hand, simple real-time duration properties between events can be verified using HyTech. However, to do so, the model must be specified by introducing auxiliary variables to measure delays between events or the delay needed for a particular conditions to be hold. Bounded response time and minimal event separation are further properties that can be verified using HyTech. These properties, however, can only be checked after augmenting the model under consideration with what is called a *monitor* or *observer* automaton (cf. [13]), whose functionality is to observe the model without changing its behavior under consideration. It records the time as soon as some event occurs. Before the model is verified, the monitor automaton has to be composed with the original model, which in turns may add further complexity to the model. As demonstrated in this paper, however, there is no need to augment the model with an extra automaton for the reason that during the run, not only the states of variables are recorded, but also the events and the time, where the constraint solver can be used to reason about the respective property

6.1 Related Works

Several tools exist for formal verification of hybrid automata [13, 8, 3], where a multi-agent team can be verified. Differently to our approach, however, these tools compose the automata prior to the verification phase.

We are not the first one who approached modeling and verifying hybrid automata using CLP. In contrast to our proposed approach, several authors propose the explicit composition of different concurrent automata by hand leading to one single automaton, before a CLP implementation is applied. This is a tedious work, especially in the case of multi-agent systems, where a group of agents exists. The latter case is exemplified in [25, 20].

Other authors employ CLP for implementing hybrid automata [4, 5, 10], but restrict their attention to a simple class of hybrid systems (e.g. timed systems). They do not construct the overall behavior prior to modeling, but model each automaton separately. However, the run of the model takes all possible paths into consideration, resulting from the product of each component, which leads to unnecessary computation.

7 Conclusion

Multi-agent systems need to coordinate their plans especially in a safety critical environment, where unexpected events typically arise. Therefore, it is becoming increasingly important to react to those events in real time in order to avoid the risk that may occur during the planning. For this purpose, various researches have approached hybrid automata as a framework to model reactively multi-agent plans. In this paper, we have showed how multi-agent systems can be formally specified and verified as hybrid automata without explicitly composing the system prior to the verification phase. The previous helps to tackle the state space problem that may arise during the composition process. We have programmed our approach by means of constraint logic programming, where constraint solvers help us to build dynamically the entire behavior of a multi-agent system and to reason about its properties. Furthermore, we have showed how various properties can be verified using our CLP framework. In addition, we have conducted several experiments taken from the hybrid automata context to show the feasibility of our approach.

CLP is a suitable framework, where we can reason not only about the time behaviors, bout also about the knowledge. The combination of both worlds is subjected to future work.

References

1. R. Alur, C. Courcoubetis, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. In *ICAOS: International Conference on Analysis and Optimization of Systems – Discrete-Event Systems*, Lecture Notes in Control and Information Sciences 1994, pages 331–351. Springer, Berlin, Heidelberg, New York, 1994.
2. R. Alur and D. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
3. G. Behrmann, A. David, and K. G. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *Proceedings of 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems – Formal Methods for the Design of Real-Time Systems (SFM-RT)*, LNCS 3185, pages 200–236. Springer, Berlin, Heidelberg, New York, 2004.
4. A. Ciarlini and T. Frühwirth. Automatic derivation of meaningful experiments for hybrid systems. *Proceeding of ACM SIGSIM Conf. on Artificial Intelligence, Simulation, and Planning (AIS'00)*, 2000.
5. G. Delzanno and A. Podelski. Model checking in CLP. *Proceeding, Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, pages 223–239, 1999.
6. M. Egerstedt. Behavior Based Robotics Using Hybrid Automata. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 103–116, 2000.
7. A. El Fallah-Seghrouchni, I. Degirmenciyan-Cartault, and F. Marc. Framework for Multi-agent Planning Based on Hybrid Automata. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 226–235, 2003.
8. G. Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. In M. Morari and L. Thiele, editors, *Hybrid Systems: Computation and Control, 8th International Workshop, Proceedings*, LNCS 3414, pages 258–273. Springer, 2005.

9. U. Furbach, J. Murray, F. Schmidsberger, and F. Stolzenburg. Hybrid multiagent systems with timed synchronization – specification and model checking. In M. Dastani, A. El Fallah Seghrouchni, A. Ricci, and M. Winikoff, editors, *Post-Proceedings of 5th International Workshop on Programming Multi-Agent Systems at 6th International Joint Conference on Autonomous Agents & Multi-Agent Systems*, LNAI 4908, pages 205–220. Springer, 2008.
10. G. Gupta and E. Pontelli. A constraint-based approach for specification and verification of real-time systems. *Proceedings of IEEE Real-time Symposium*, pages 230–239, 1997.
11. N. Halbwachs, Y. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In *Static Analysis – Proceedings of 1st International Static Analysis Symposium (SAS'94)*, LNCS 864, pages 223–223, Namur, Belgium, 1994. Springer, Berlin, Heidelberg, New York.
12. T. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 278–292, New Brunswick, NJ, 1996. IEEE Computer Society Press.
13. T. Henzinger, P.-H. Ho, and H. Wong-Toi. A user guide to HyTech. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 1019, pages 41–71. Springer, Berlin, Heidelberg, New York, 1995.
14. T. Henzinger, B. Horowitz, R. Majumdar, and H. Wong-Toi. Beyond HYTECH: Hybrid Systems Analysis Using Interval Numerical Methods. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 130–144, 2000.
15. T. Henzinger, P. Kopke, A. Puri, and P. Varaiya. What's Decidable about Hybrid Automata? *Journal of Computer and System Sciences*, 57(1):94–124, 1998.
16. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: The Next Generation. In *IEEE Real-Time Systems Symposium*, pages 56–65, 1995.
17. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *IEEE Transactions on Automatic Control*, 43:540–554, 1998.
18. G. Hutzler, H. Klaudel, and D. Y. Wang. Towards timed automata and multi-agent systems. In *Formal Approaches to Agent-Based Systems, Third InternationalWorkshop, FAABS 2004, Greenbelt, MD, USA, April 26-27, 2004, Revised Selected Papers*, volume 3228 of *Lecture Notes in Computer Science*, pages 161–172. Springer, 2005.
19. J. Jaffar and J. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119. ACM New York, NY, USA, 1987.
20. J. Jaffar, A. Santosa, and R. Voicu. A clp proof method for timed automata. *Real-Time Systems Symposium, IEEE International*, 0:175–186, 2004.
21. M. W. Krzysztof R. Apt. *Constraint Logic Programming Using Eclipse*. Cambridge University Press, Cambridge, UK, 2007.
22. A. Mohammed and U. Furbach. Modeling multi-agent logistic process system using hybrid automata. In U. Ultes-Nitsche, D. Moldt, and J. C. Augusto, editors, *MSVVEIS 2008: Proceedings of the 6th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, MSVVEIS-2008*, pages 141–149. INSTICC PRESS, 2008.
23. A. Mohammed and F. Stolzenburg. Implementing hierarchical hybrid automata using constraint logic programming. In S. Schwarz, editor, *Proceedings of 22nd Workshop on (Constraint) Logic Programming*, pages 60–71, Dresden, 2008. University Halle Wittenberg, Institute of Computer Science. Technical Report 2008/08.
24. Object Management Group, Inc. *UML Version 2.1.2 (Infrastructure and Superstructure)*, November 2007.
25. L. Urbina. Analysis of hybrid systems in CLP(R). In *Proceedings of 2nd International Conference on Principles and Practice of Constraint Programming (CP'96)*, LNAI 1118, pages 451–467, 1996.

Probabilistic Behavioural State Machines

Peter Novák

Department of Informatics
Clausthal University of Technology
Julius-Albert-Str. 4, D-38678 Clausthal-Zellerfeld, Germany
peter.novak@tu-clausthal.de

Abstract Development of embodied cognitive agents in agent oriented programming languages naturally leads to writing underspecified programs. The semantics of BDI inspired rule based agent programming languages leaves room for various alternatives as to how to implement the action selection mechanism of an agent (paraphrased from [5]).
To facilitate encoding of heuristics for the non-deterministic action selection mechanism, I introduce a probabilistic extension of the framework of *Behavioural State Machines* and its associated programming language interpreter *Jazzyk*. The language rules coupling a triggering condition and an applicable behaviour are extended with labels, thus allowing finer grained control of the behaviour selection mechanism of the underlying interpreter. In consequence, the agent program not only prescribes a set of mental state transitions enabled in a given context, but also specifies a probability distribution over them.

1 Introduction

Situated cognitive agents, such as mobile service robots, operate in rich, unstructured, dynamically changing and not completely observable environments. Since various phenomena of real world environments are not completely specifiable, as well as because of limited, noisy, or even malfunctioning sensors and actuators, such agents must operate with *incomplete information*.

On the other hand, similarly to mainstream software engineering, *robustness* and *elaboration tolerance* are some of the desired properties for cognitive agent programs. Embodied agent is supposed to operate reasonably well also in conditions previously unforeseen by the designer and it should degrade gracefully in the face of partial failures and unexpected circumstances (robustness). At the same time the program should be concise, easily maintainable and extensible (elaboration tolerance).

Agent programs in the reactive planning paradigm [15] are specifications of partial plans for the agent about how to deal with various situations and events occurring in the environment. The inherent incomplete information on one side, stemming from a level of knowledge representation granularity chosen at the agent's design phase, and striving for robust and easily maintainable programs on the other yield a trade-off of *intentional underspecification* of resulting agent programs.

Most BDI inspired agent oriented programming languages on both sides of the spectrum between theoretically founded (such as *AgentSpeak(L)/Jason* [2], *3APL* [3] or *GOAL* [4]) to pragmatic ones (e.g., *JACK* [16] or *Jadex* [14]) facilitate encoding of underspecified, non-deterministic programs. Any given situation, or an event can at the same time trigger multiple behaviours, which themselves can be non-deterministic, i.e, can include alternative branches.

A precise and exclusive qualitative specification of behaviour triggering conditions is often impossible due to the, at the design time chosen and fixed, level of knowledge representation granularity. This renders the qualitative condition description a rather coarse grained means for steering agent's life-cycle. In such contexts, a quantitative heuristics steering the language interpreter's choices becomes a powerful tool for encoding developer's informal knowledge, or intuitions about agent's run-time evolutions. For example, it might be appropriate to execute some applicable behaviours more often than others, or some of them might intuitively perform better than other behaviours in the same context, and therefore should be preferably selected.

In this paper I propose a *probabilistic extension* of a rule-based agent programming language. The core idea is straightforward: language rules coupling a triggering condition with an applicable behaviour are extended with labels denoting a probability with which the interpreter's selection mechanism should choose the particular behaviour in a given context. The idea is directly applicable also to other agent programming languages, however here I focus on extension of the theoretical framework of *Behavioural State Machines* [10] and its associated programming language instance *Jazzyk*, which I use in my long-term research. One of elegant implications of the extension of the *BSM* framework is that subprograms with labelled rules can be seen as specifications of probability distributions over actions applicable in a given context. This allows steering agent's focus of deliberation on a certain sub-behaviour with only minor changes to the original agent program. I call this technique *adjustable deliberation*.

After a brief overview of the framework of *Behavioural State Machines* (*BSM*) with the associated programming language *Jazzyk* in Section 2, sections 3 and 4 introduce *P-BSM* and *Jazzyk(P)*, their respective probabilistic extensions. Section 5 discusses practical use of the *P-BSM* framework together with a brief overview of related work. Finally, a summary with final remarks concludes the paper in Section 6.

2 Behavioural State Machines

In [10] I introduced the framework of *Behavioural State Machines*. *BSM* framework draws a clear distinction between the *knowledge representation* and *behavioural* layers within an agent. It thus provides a programming framework that clearly separates the programming concerns of *how to represent an agent's knowledge* about, for example, its environment and *how to encode its behaviours* for acting in it. This section briefly introduces the *BSM* framework, for simplic-

ity without treatment of variables. For the complete formal description of the *BSM* framework, see [10].

2.1 Syntax

BSM agents are collections of one or more so-called *knowledge representation modules* (KR modules), typically denoted by $\mathcal{M}$, each representing a part of the agent's knowledge base. KR modules may be used to represent and maintain various mental attitudes of an agent, such as knowledge about its environment, or its goals, intentions, obligations, etc. Transitions between states of a *BSM* result from applying so-called *mental state transformers* (*mst*), typically denoted by τ. Various types of mst's determine the behaviour that an agent can generate. A *BSM agent* consists of a set of KR modules $\mathcal{M}_1, \ldots, \mathcal{M}_n$ and a mental state transformer $\mathcal{P}$, i.e. $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P})$; the mst $\mathcal{P}$ is also called an *agent program*.

The notion of a KR module is an abstraction of a partial knowledge base of an agent. In turn, its states are to be treated as theories (i.e., sets of sentences) expressed in the KR language of the module. Formally, a KR module $\mathcal{M}_i = (\mathcal{S}_i, \mathcal{L}_i, \mathcal{Q}_i, \mathcal{U}_i)$ is characterised by a knowledge representation language $\mathcal{L}_i$, a set of states $\mathcal{S}_i \subseteq 2^{\mathcal{L}_i}$, a set of query operators $\mathcal{Q}_i$ and a set of update operators $\mathcal{U}_i$. A query operator $\models \in \mathcal{Q}_i$ is a mapping $\models : \mathcal{S}_i \times \mathcal{L}_i \rightarrow \{\top, \bot\}$. Similarly an update operator $\oplus \in \mathcal{U}_i$ is a mapping $\oplus : \mathcal{S}_i \times \mathcal{L}_i \rightarrow \mathcal{S}_i$.

Queries, typically denoted by φ, can be seen as operators of type $\models : \mathcal{S}_i \rightarrow \{\top, \bot\}$. A primitive query $\varphi = (\models \phi)$ consists of a query operator $\models \in \mathcal{Q}_i$ and a formula $\phi \in \mathcal{L}_i$ of the same KR module $\mathcal{M}_i$. Complex queries can be composed by means of conjunction $\wedge$, disjunction $\vee$ and negation $\neg$.

Mental state transformers enable transitions from one state to another. A primitive mst $\oslash\psi$, typically denoted by ρ and constructed from an update operator $\oslash \in \mathcal{U}_i$ and a formula $\psi \in \mathcal{L}_i$, refers to an update on the state of the corresponding KR module. Conditional mst's are of the form $\varphi \longrightarrow \tau$, where φ is a query and τ is a mst. Such a conditional mst makes the application of τ depend on the evaluation of φ. Syntactic constructs for combining mst's are: non-deterministic choice $|$ and sequence $\circ$.

Definition 1 (mental state transformer). *Let $\mathcal{M}_1, \ldots, \mathcal{M}_n$ be KR modules of the form $\mathcal{M}_i = (\mathcal{S}_i, \mathcal{L}_i, \mathcal{Q}_i, \mathcal{U}_i)$. The set of* mental state transformers *is defined as below:*

- **skip** *is a* primitive *mst,*
- *if $\oslash \in \mathcal{U}_i$ and $\psi \in \mathcal{L}_i$, then $\oslash\psi$ is a* primitive *mst,*
- *if φ is a query, and τ is a mst, then $\varphi \longrightarrow \tau$ is a* conditional *mst,*
- *if τ and τ' are mst's, then $\tau|\tau'$ and $\tau \circ \tau'$ are mst's (*choice*, and* sequence *respectively).*

2.2 Semantics

The *yields* calculus, summarised below after [10], specifies an update associated with executing a mental state transformer in a single step of the language inter-

preter. It formally defines the meaning of the state transformation induced by executing an mst in a state, i.e., a mental state transition.

Formally, a *mental state* σ of a *BSM* $\mathcal{A} = (\mathcal{M}_1, \dots, \mathcal{M}_n, \tau)$ is a tuple $\sigma = \langle \sigma_1, \dots, \sigma_n \rangle$ of KR module states $\sigma_1 \in \mathcal{S}_1, \dots, \sigma_n \in \mathcal{S}_n$, corresponding to $\mathcal{M}_1, \dots, \mathcal{M}_n$ respectively. $\mathcal{S} = \mathcal{S}_1 \times \cdots \times \mathcal{S}_n$ denotes the space of all mental states over $\mathcal{A}$. A mental state can be modified by applying primitive mst's on it and query formulae can be evaluated against it. The semantic notion of truth of a query is defined through the satisfaction relation $\models$. A primitive query $\vDash\phi$ holds in a mental state $\sigma = \langle \sigma_1, \dots, \sigma_n \rangle$ (written $\sigma \models (\vDash\phi)$) iff $\vDash(\phi, \sigma_i)$, otherwise we have $\sigma \not\models (\vDash\phi)$. Given the usual meaning of Boolean operators, it is straightforward to extend the query evaluation to compound query formulae. Note that evaluation of a query does not change the mental state σ.

For an mst $\oslash\psi$, we use $(\oslash, \psi)$ to denote its semantic counterpart, i.e., the corresponding *update* (state transformation). Sequential application of updates is denoted by $\bullet$, i.e. $\rho_1 \bullet \rho_2$ is an update resulting from applying ρ_1 first and then applying ρ_2. The application of an update to a mental state is defined formally below.

Definition 2 (applying an update). *The result of applying an update* $\rho = (\oslash, \psi)$ *to a state* $\sigma = \langle \sigma_1, \dots, \sigma_n \rangle$ *of a* BSM $\mathcal{A} = (\mathcal{M}_1, \dots, \mathcal{M}_n, \mathcal{P})$*, denoted by* $s \bigoplus \rho$*, is a new state* $\sigma' = \langle \sigma_1, \dots, \sigma'_i, \dots, \sigma_n \rangle$*, where* $\sigma'_i = \sigma_i \oslash \psi$ *and* σ_i*,* $\oslash$*, and* ψ *correspond to one and the same* $\mathcal{M}_i$ *of* $\mathcal{A}$*. Applying the empty update* **skip** *on the state* σ *does not change the state, i.e.* $\sigma \bigoplus \mathbf{skip} = \sigma$*.*

Inductively, the result of applying a sequence of updates $\rho_1 \bullet \rho_2$ *is a new state* $\sigma'' = \sigma' \bigoplus \rho_2$*, where* $\sigma' = \sigma \bigoplus \rho_1$*.* $\sigma \overset{\rho_1 \bullet \rho_2}{\to} \sigma'' = \sigma \overset{\rho_1}{\to} \sigma' \overset{\rho_2}{\to} \sigma''$ *denotes the corresponding compound transition.*

The meaning of a mental state transformer in state σ, formally defined by the *yields* predicate below, is the update set it yields in that mental state.

Definition 3 (yields calculus). *A mental state transformer* τ *yields an* update ρ *in a state* σ*, iff* $yields(\tau, \sigma, \rho)$ *is derivable in the following calculus:*

$$\frac{\top}{yields(\mathbf{skip},\sigma,\mathbf{skip})} \quad \frac{\top}{yields(\oslash\psi,\sigma,(\oslash,\psi))} \qquad \textit{(primitive)}$$

$$\frac{yields(\tau,\sigma,\rho),\ \sigma\models\phi}{yields(\phi\longrightarrow\tau,\sigma,\rho)} \quad \frac{yields(\tau,\sigma,\rho),\ \sigma\not\models\phi}{yields(\phi\longrightarrow\tau,\sigma,\mathbf{skip})} \qquad \textit{(conditional)}$$

$$\frac{yields(\tau_1,\sigma,\rho_1),\ yields(\tau_2,\sigma,\rho_2)}{yields(\tau_1|\tau_2,\sigma,\rho_1),\ yields(\tau_1|\tau_2,\sigma,\rho_2)} \qquad \textit{(choice)}$$

$$\frac{yields(\tau_1,\sigma,\rho_1),\ yields(\tau_2,\sigma \bigoplus \rho_1,\rho_2)}{yields(\tau_1\circ\tau_2,\sigma,\rho_1\bullet\rho_2)} \qquad \textit{(sequence)}$$

We say that τ *yields an* update set ν *in a state* σ *iff* $\nu = \{\rho | yields(\tau, \sigma, \rho)\}$*.*

The mst **skip** yields the update **skip**. Similarly, a primitive update mst $\oslash\psi$ yields the corresponding update $(\oslash, \psi)$. In the case the condition of a conditional mst $\phi \longrightarrow \tau$ is satisfied in the current mental state, the calculus yields one of the updates corresponding to the right hand side mst τ, otherwise the no-operation

skip update is yielded. A non-deterministic choice mst yields an update corresponding to either of its members and finally a sequential mst yields a sequence of updates corresponding to the first mst of the sequence and an update yielded by the second member of the sequence in a state resulting from application of the first update to the current mental state.

Notice, that the provided semantics of choice and sequence operators implies associativity of both. Hence, from this point on, instead of the strictly pairwise notation $\tau_1|(\tau_2|(\tau_3|(\cdots|\tau_k)))$, we simply write $\tau_1|\tau_2|\tau_2|\cdots|\tau_k$. Similarly for the sequence operation $\circ$.

The following definition articulates the denotational semantics of the notion of mental state transformer as an encoding of a function mapping mental states of a *BSM* to updates, i.e., transitions between them.

Definition 4 (mst functional semantics). *Let $\mathcal{M}_1, \ldots, \mathcal{M}_n$ be KR modules. A mental state transformer τ encodes a function $\mathfrak{f}_\tau : \sigma \mapsto \{\rho | yields(\tau, \sigma, \rho)\}$ over the space of mental states $\sigma = \langle \sigma_1, \ldots, \sigma_n \rangle \in S_1 \times \cdots \times S_n$.*

Subsequently, the semantics of a *BSM* agent is defined as a set of traces in the induced transition system enabled by the *BSM* agent program.

Definition 5 (BSM semantics). *A* BSM *$\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P})$ can make a step from state σ to a state σ', iff $\sigma' = \sigma \bigoplus \rho$, s.t. $\rho \in \mathfrak{f}_\mathcal{P}(\sigma)$. We also say, that* $\mathcal{A}$ induces *a (possibly compound) transition $\sigma \xrightarrow{\rho} \sigma'$.*

A possibly infinite sequence of states $\sigma_1, \ldots, \sigma_i, \ldots$ is a run *of BSM $\mathcal{A}$, iff for each $i \geq 1$, $\mathcal{A}$ induces a transition $\sigma_i \rightarrow \sigma_{i+1}$.*

The semantics of an agent system characterised by a BSM $\mathcal{A}$, is a set of all runs of $\mathcal{A}$.

Additionally, we require the non-deterministic choice of a *BSM* interpreter to fulfil the *weak fairness condition*, similar to that in [7], for all the induced runs.

Condition 1 (weak fairness condition) *A computation run is weakly fair iff it is not the case that an update is always yielded from some point in time on but is never selected for execution.*

2.3 Jazzyk

Jazzyk is an interpreter of the *Jazzyk* programming language implementing the computational model of the *BSM* framework. Later in this paper, we use a more readable notation mixing the syntax of *Jazzyk* with that of the *BSM* mst's introduced above. when ϕ then τ encodes a conditional mst $\phi \longrightarrow \tau$. Symbols ; and , stand for choice | and sequence $\circ$ operators respectively. To facilitate operator precedence, mental state transformers can be grouped into compound structures, blocks, using curly braces {...}.

To better support source code modularity and re-usability, *Jazzyk* interpreter integrates a macro preprocessor, a powerful tool for structuring and modularising and encapsulating the source code and writing code templates.

For further details on the *Jazzyk* programming language and the macro preprocessor integration with *Jazzyk* interpreter, consult [10].

3 Probabilistic BSMs

In the plain *BSM* framework, the syntactic construct of a mental state transformer encodes a transition function over the space of mental states of a *BSM* (cf. Definition 4). Hence, an execution of a compound non-deterministic choice mst amounts to a non-deterministic selection of one of its components and its subsequent application to the current mental state of the agent. In order to enable a finer grained control over this selection process, in this section I introduce an extension of the *BSM* framework with specifications of a probability distributions over components of choice mst's.

The *P-BSM* formalism introduced below heavily builds on associativeness of *BSM* composition operators of non-deterministic choice and sequence. We also informally say that an mst τ *occurs* in a mst τ' iff τ' can be constructed from a set of mst's $\mathcal{T}$, s.t. $\tau \in \mathcal{T}$, by using composition operators as defined by the Definition 1.

Definition 6 (Probabilistic BSM). *A* Probabilistic Behavioural State Machine *(*P-BSM*)* $\mathcal{A}_p$ *is a tuple* $\mathcal{A}_p = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P}, \Pi)$*, where* $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P})$ *is a* BSM *and* $\Pi : \tau \mapsto P_\tau$ *is a function assigning to each non-deterministic choice mst of the form* $\tau = \tau_1|\cdots|\tau_k \in \mathcal{P}$ *occurring in* $\mathcal{P}$ *a discrete probability distribution function* $P_\tau{:}\tau_i \mapsto [0,1]$*, s.t.* $\sum_{i=1}^{k} P_\tau(\tau_i) = 1$*.*

W.l.o.g. we assume that each mst occurring in the agent program $\mathcal{P}$ *can be uniquely identified (e.g. by its position in the agent program).*

The probability distribution function P_τ assigns to each component of a non-deterministic choice mst $\tau = \tau_1|\tau_2|\cdots|\tau_k$ a probability of its selection for application by a *BSM* interpreter.

Note, that because of the unique identification of mst's in an agent program $\mathcal{P}$, the function Π assigns two distinct discrete probability distributions P_{τ_1} and P_{τ_2} to choice mst's τ_1, τ_2 even when they share the syntactic form but occur as distinct components of $\mathcal{P}$.

To distinguish from the *BSM* formalism, we call mst's occurring in a *P-BSM probabilistic mental state transformers*. *BSM* mst's as defined in Section 2 will be called *plain*.

Similarly to plain mst's, the semantic counterpart of a probabilistic mst is a probabilistic update. A *probabilistic update* of a *P-BSM* $\mathcal{A}_p = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P}, \Pi)$ is a tuple $p{:}\rho$, where $p \in \mathbb{R}$, s.t. $p \in [0,1]$, is a probability and $\rho = (\oslash, \psi)$ is an update from the *BSM* $\mathcal{A} = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P})$.

The semantics of a probabilistic mental state transformer in a state σ, formally defined by the $yields_p$ predicate below, is the probabilistic update set it yields in that mental state.

Definition 7 (yields$_p$ calculus). *A probabilistic mental state transformer* τ *yields a probabilistic update* $p{:}\rho$ *in a state* σ*, iff* $yields_p(\tau, \sigma, p{:}\rho)$ *is derivable in the following calculus:*

$$\frac{\top}{\mathit{yields}_p(\mathbf{skip},\sigma,1{:}\mathbf{skip})} \qquad \frac{\top}{\mathit{yields}_p(\oslash\psi,\sigma,1{:}(\oslash,\psi))} \qquad \textit{(primitive)}$$

$$\frac{\mathit{yields}_p(\tau,\sigma,p{:}\rho),\ \sigma\models\phi}{\mathit{yields}_p(\phi\longrightarrow\tau,\sigma,p{:}\rho)} \qquad \frac{\mathit{yields}_p(\tau,\sigma,\theta,p{:}\rho),\ \sigma\not\models\phi}{\mathit{yields}_p(\phi\longrightarrow\tau_p,\sigma,1{:}\mathbf{skip})} \qquad \textit{(conditional)}$$

$$\frac{\tau=\tau_1|\cdots|\tau_k,\ \Pi(\tau)=P_\tau,\ \forall 1\leq i\leq k{:}\ \mathit{yields}_p(\tau_i,\sigma,p_i{:}\rho_i)}{\forall 1\leq i\leq k{:}\ \mathit{yields}_p(\tau,\sigma,P_\tau(\tau_i)\cdot p_i{:}\rho_i)} \qquad \textit{(choice)}$$

$$\frac{\tau=\tau_1\circ\cdots\circ\tau_k,\ \forall 1\leq i\leq k{:}\ \mathit{yields}_p(\tau_i,\sigma_i,p_i{:}\rho_i)\wedge\sigma_{i+1}=\sigma_i\bigoplus\rho_i}{\mathit{yields}(\tau,\sigma_1,\prod_{i=1}^{k}p_i{:}\rho_1\bullet\cdots\bullet\rho_k)} \qquad \textit{(sequence)}$$

The modification of the plain *BSM yields* calculus introduced above for primitive and conditional mst's is rather straightforward. A plain primitive mst yields the associated primitive update for which there's no probability of execution specified. A conditional mst yields probabilistic updates of its right hand side if the left hand side query condition is satisfied. It amounts to a **skip** mst otherwise. The function Π associates a discrete probability distribution function with each non-deterministic choice mst and thus modifies the probability of application of the probabilistic updates yielded by its components accordingly. Finally, similarly to the plain *yields* calculus, a sequence of probabilistic mst's yields sequences of updates of its components, however the joint application probability equals to the conditional probability of selecting the particular sequence of updates. The following example illustrates the sequence rule of the probabilistic yields_p calculus.

Example 1. Consider the following mst: $(0.3{:}\tau_1 \,|\, 0.7{:}\tau_2) \circ (0.6{:}\tau_3 \,|\, 0.4{:}\tau_4)$. Let's assume that for each of the component mst's τ_i, we have $\mathit{yields}_p(\tau_i, \sigma, p_i{:}\rho_i)$ in a state σ. The plain *yields* calculus yields the following sequences of updates $\rho_1 \bullet \rho_3$, $\rho_1 \bullet \rho_4$, $\rho_2 \bullet \rho_3$ and $\rho_2 \bullet \rho_4$. The probability of selection of each of them, however, equals to the conditional probability of choosing an update from the second component of the sequence, provided that the choice from the first one was already made. I.e. the probabilistic yields_p calculus results in the following sequences of probabilistic updates $0.18{:}(\rho_1 \bullet \rho_3)$, $0.12{:}(\rho_1 \bullet \rho_4)$, $0.42{:}(\rho_2 \bullet \rho_3)$ and $0.28{:}(\rho_2 \bullet \rho_4)$.

The corresponding adaptation of the mst functional semantics straightforwardly follows.

Definition 8 (probabilistic mst functional semantics). *Let* $\mathcal{A}_p = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P}, \Pi)$ *be a* P-BSM. *A probabilistic mental state transformer* τ *encodes a transition function* $\mathfrak{fp}_\tau : \sigma \mapsto \{p : \rho | \mathit{yields}_p(\tau, \sigma, p : \rho)\}$ *over the space of mental states* $\sigma = \langle\sigma_1, \ldots, \sigma_n\rangle \in S_1 \times \cdots \times S_n$.

According to the Definition 6, each mst occurring in a *P-BSM* agent program can be uniquely identified. Consequently, also each probabilistic update yielded by the program can be uniquely identified by the mst it corresponds to. The consequence is, that w.l.o.g. we can assume that even when two probabilistic updates $p_1{:}\rho_1$, $p_2{:}\rho_2$ yielded by the agent program $\mathcal{P}$ in a state σ share their syntactic form (i.e. $p_1 = p_2$ and ρ_1, ρ_2 encode the same plain *BSM* update) they both independently occur in the probabilistic update set $\mathfrak{fp}(\sigma)$.

The following lemma shows, that the semantics of probabilistic mst's embodied by the $yields_p$ calculus can be understood as *an encoding of a probability distribution*, or *a probabilistic policy* over updates yielded by the underlying plain mst. Moreover, it also implies that composition of probabilistic mst's maintains their nature as probability distributions.

Lemma 1. *Let* $\mathcal{A}_p = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P}, \Pi)$ *be a* P-BSM. *For every mental state transformer* τ *occurring in* $\mathcal{P}$ *and a mental state* σ *of* $\mathcal{A}_p$*, we have*

$$\sum_{p:\rho \in \mathfrak{fp}_\tau(\sigma)} p = 1 \tag{1}$$

Proof. Cf. Appendix A.

Finally, the semantics of a *P-BSM* agent is defined as a set of traces in the induced transition system enabled by the *P-BSM* agent program.

Definition 9 (BSM semantics). *A* P-BSM $\mathcal{A}_p = (\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{P}, \Pi)$ *can make a step from state* σ *to a state* σ' *with probability* p*, iff* $\sigma' = \sigma \bigoplus \rho$*, s.t.* $p{:}\rho \in \mathfrak{fp}_\tau(\sigma)$*. We also say, that with a probability* p*,* $\mathcal{A}_p$ *induces a (possibly compound) transition* $\sigma \stackrel{p:\rho}{\to} \sigma'$*.*

A possibly infinite sequence of states $\omega = \sigma_1, \ldots, \sigma_i, \ldots$ *is a run of* P-BSM $\mathcal{A}_p$*, iff for each* $i \geq 1$*,* $\mathcal{A}$ *induces the transition* $\sigma_i \stackrel{p_i:\rho_i}{\to} \sigma_{i+1}$ *with probability* p_i*.*

Let $pref(\omega)$ *denote the set of all finite prefixes of a possibly infinite computation run* ω *and* $|.|$ *the length of a finite run.* $P(\omega) = \prod_{i=1}^{|\omega|} p_i$ *is then the probability of the finite run* ω*.*

The semantics of an agent system characterised by a P-BSM $\mathcal{A}_p$*, is a set of all runs* ω *of* $\mathcal{A}_p$*, s.t. all of their finite prefixes* $\omega' \in pref(\omega)$ *have probability* $P(\omega') > 0$*.*

Informally, the semantics of an agent system is a set of runs involving only transitions induced by updates with a non-zero selection probability.

Additionally, we require an admissible *P-BSM* interpreter to fulfil the following specialisation of the weak fairness condition, for all the induced runs.

Condition 2 (P-BSM weak fairness condition) *Let* ω *be a possibly infinite computation run of a* P-BSM $\mathcal{A}_p$*. Let also* $freq_{p:\rho}(\omega')$ *be the number of transitions induced by the update* $p{:}\rho$ *along a finite prefix of* $\omega' \in pref(\omega)$*.*

We say that ω *is* weakly fair w.r.t. $\mathcal{A}_p$ *iff for all updates* $p{:}\rho$ *we have, that if from some point on* $p{:}\rho$ *is always yielded in states along* ω*, 1) it also occurs on* ω *infinitely often, and 2) for the sequence of finite prefixes of* ω *ordered according to their length holds*

$$\liminf_{\substack{|\omega'| \to \infty \\ \omega' \in pref(\omega)}} \frac{freq_{p:\rho}(\omega')}{|\omega'|} \geq p$$

Similarly to the plain *BSM* weak fairness Condition 1, the above stated Condition 2 embodies a minimal requirement on admissible *P-BSM* interpreters. It admits only *P-BSM* interpreters which honor the intended probabilistic semantics of the non-deterministic choice selection of the $yields_p$ calculus. The first part of the requirement is a consequence of the plain *BSM* weak fairness condition (Condition 1), while the second states that in sufficiently long computation runs, the frequency of occurrence of an always yielded probabilistic update corresponds to its selection probability in each single step.

4 Jazzyk(P)

Jazzyk is a programming language instantiating the plain *BSM* theoretical framework introduced in [10]. This section informally describes its extension *Jazzyk(P)*, an instantiation of the framework of *Probabilistic Behavioural State Machines* introduced in Section 3 above.

Jazzyk(P) syntax differs from that of *Jazzyk* only in specification of probability distributions over choice mst's. *Jazzyk(P)* allows for explicit labellings of choice mst members by their individual application probabilities. Consider the following labelled choice mst $p_1{:}\tau_1 \,;\, p_2{:}\tau_2 \,;\, p_3{:}\tau_3 \,;\, p_4{:}\tau_4$ in the *Jazzyk(P)* notation. Each $p_i \in [0,1]$ denotes the probability of selection of mst τ_i by the interpreter. Furthermore, to ensure that the labelling denotes a probability distribution over τ_i's, *Jazzyk(P)* parser requires that $\sum_{i=1}^{k} p_i = 1$ for every choice mst $p_1{:}\tau_1 \,;\ldots;\, p_k{:}\tau_k$ occurring in the considered agent program. Similarly to *Jazzyk*, during the program interpretation phase, *Jazzyk(P)* interpreter proceeds in a top-down manner subsequently considering nested mst's from the main agent program, finally down to primitive update formulae. When the original *Jazzyk* interpreter faces a selection from a non-deterministic choice mst, it randomly selects one of them assuming a discrete uniform probability distribution. I.e., the probability of selecting from a choice mst with k members is $\frac{1}{k}$ for each of them. The extended interpreter *Jazzyk(P)* respects the specified selection probabilities: it generates a random number $p \in [0,1]$ and selects τ_s, s.t. $\sum_{i=1}^{s-1} p_i \leq p \leq \sum_{i=1}^{s} p_i$.

For convenience, *Jazzyk(P)* enables use of incomplete labellings. An *incompletely labelled* non-deterministic choice mst is one containing at least one member mst without an explicit probability specification such as $p_1{:}\tau_1 \,;\, p_2{:}\tau_2 \,;\, \tau_3 \,;\, \tau_4$. In such a case, the *Jazzyk(P)* parser automatically completes the distribution by uniformly dividing the remaining probability range to unlabelled mst's. I.e., provided an incompletely labelled choice mst with k members, out of which $m < k$ are labelled $(p_1{:}\tau_1 \,;\ldots;\, p_m{:}\tau_m \,;\, \tau_{m+1} \,;\cdots;\, \tau_k)$, it assigns probability $p = \frac{1-\sum_{i=1}^{m} p_i}{k-m}$ to the remaining mst's $\tau_{m+1}, \ldots, \tau_k$.

The Listing 1 provides an example of a *Jazzyk(P)* code snippet adapted from the *Jazzbot* project [6]. Consider a BDI-style virtual agent (bot) in a simulated 3D environment. The bot moves around a virtual building and searches for items which it picks up and delivers to a particular place in the environment. Upon encountering an unfriendly agent (attacker), it executes an emergency behaviour,

Listing 1 Example of *Jazzyk(P)* syntax.

```
when ⊨bel [{ threatened }] then {
        /* ***Emergency modus operandi*** */

        /* Detect the enemy's position */
    0.7 : when ⊨bel [{ attacker(Id) }] and ⊨env [{ eye see Id player Pos }]
    then ⊕map [{ positions[Id] = Pos }] ;

    /* Check the camera sensor */
    0.2 : when ⊨env [{ eye see Id Type Pos }] then {
        ⊕bel [{ see(Id, Type) }] ,
        ⊕map [{ objects[Pos].addIfNotPresent(Id) }]
    }

    /* Check the body health sensor */
    when ⊨env [{ body health X }] then ⊕bel [{ health(X). }] ;
} else {
    /* ***Normal mode of perception*** */

    /* Check the body health sensor */
    when ⊨env [{ body health X }] then ⊕bel [{ health(X). }] ;

    /* Check the camera sensor */
    when ⊨env [{ eye see Id Type Pos }] then {
        ⊕bel [{ see(Id, Type) }] ,
        ⊕map [{ positions[Id] = Pos }]
    }
}
```

such as running away until it feels safe again. The agent consists of several KR modules *bel*, *map* and *env* respectively representing its beliefs about the environment and itself, the map of the environment and an interface to its sensors and actuators, i.e. the body. The corresponding query and update operators $\models$ and $\oplus$ are sub-scripted with the KR module label they correspond to.

The Listing 1 provides a piece of code for perception of the bot. In the normal mode of operation, the bot in a single step queries either its camera, or its body health status sensor with the same probability of selection for each of them, i.e., 0.5. However, in the case of emergency, the bot focuses more on escaping the attacker, therefore, in order to retrieve the attacker's position, it queries the camera sensor more often (selection probability $p = 0.7$) than sensing objects around it ($p = 0.2$). Checking it's own body health is of the least importance ($p = 0.1$), however not completely negligible.

In an implemented program, however, the Listing 1 would be rewritten using the macro facility of the *Jazzyk* interpreter and reduced to a more concise code shown in the Listing 2.

5 Discussion

Probabilistic Behavioural State Machines, and in turn *Jazzyk(P)*, allow for labelling of alternatives in non-deterministic choice mental state transformers, thus providing a specification of a probability distribution over the set of enabled transitions for the next step in agent's life-cycle. Besides the, in the field

Listing 2 Example of focusing bot's attention during emergency situations rewritten with reusable macros.

```
when ⊨_bel [{ threatened }] then {
    /* ***Emergency modus operandi*** */
    0.7 : DETECT_ENEMY_POSITION ;
    0.2 : SENSE_CAMERA ;
        SENSE_HEALTH
} else {
    /* ***Normal mode of perception*** */
    SENSE_HEALTH ;
    SENSE_CAMERA
}
```

of rule based agent programming languages conventional, Plotkin style operational semantics [13], the *BSM* semantics allows a functional view on mental state transformers (cf. Definition 8). In turn, the informal reading of a *P-BSM* choice mst's can be seen as a specification of the probability with which the next transition will be chosen from the function denoted by the particular member mst. In other words, *a probability of applying the member mst function to the current mental state*.

The proposed extension allows a finer grained steering of the interpreter's non-deterministic selection mechanism and has applications across several niches of methodologies for rule based agent oriented programming languages. Our analyses and first experiments in the context of the *Jazzbot* application have shown that labelling probabilistic mst's is a useful means to contextually focus agent's perception (cf. Listing 2). Similarly, the labelling technique is useful in contexts, when it is necessary to execute certain behaviours with a certain given frequency. For example, approximately in about every 5th step broadcast a ping message to peers of the agent's team. Finally, the technique can be used when the agent developer has an informal intuition that preferring more frequent execution of certain behaviours over others (e.g. cheaper, but less efficient over resource intensive, but rather powerful) might suffice, or even perform better in a given context. Of course writing such programs makes sense only when a rigorous analysis of the situation is impossible, or undesirable and at the same time a suboptimal performance of the agent system is acceptable.

An instance of the latter technique for modifying the main control cycle of the agent program is what I call *adjustable deliberation*. Consider the following *Jazzyk BSM* code for the main control cycle of an agent adapted from [12]:

```
PERCEIVE ; HANDLE_GOALS ; ACT
```

The macros PERCEIVE, HANDLE_GOALS and ACT encode behaviours for perception (similar to that in the Listing 1), goal commitment strategy implementation and action selection respectively. In the case of emergency, as described in the example in Section 4 above, it might be useful to slightly neglect deliberation about agent's goals, in favour of an intensive environment observation and quick reaction selection. The following reformulation of the agent's control cycle demonstrates the simple program modification:

```
when ⊨_bel [{ emergency }] then { PERCEIVE ; HANDLE_GOALS ; ACT }
                           else { 0.4 : PERCEIVE ; HANDLE_GOALS ; 0.4 : ACT }
```

The underlying semantic model of *Behavioural State Machines* framework is a labelled transition system [11]. In consequence, the underlying semantic model of the *P-BSM* framework is a discrete probabilistic labelled transition system, i.e., a structure similar to a *Markov chain* [8]. This similarity suggest a relationship of the *P-BSM* underlying semantic structure to various types of *Markov models* (cf. e.g. [9]), however a more extensive exploration of this relationship is beyond the scope of this paper.

In the field of agent oriented programming languages, recently Hindriks et al. [5] introduced an extension of the *GOAL* language [4], where a quantitative numeric value is associated with execution of an action leading from a mental state m to another mental state m'. I.e., a triple of a precondition ϕ (partially describing m), an action a and a post-condition ψ (describing m') is labelled with a utility value $U(\phi, a, \psi)$. Subsequently, in each deliberation cycle, the interpreter selects the action with the highest expected future utility w.r.t. agent's goals.

The approach of Hindriks et al. focuses on estimating aggregate utility values of bounded future evolutions of the agent system, i.e., evaluating possible future courses of evolution, plans, the agent can consider, and subsequently choosing an action advancing the system evolution along the best path. The *P-BSM*, on the other hand, is concerned only with selection of the next action resulting from the bottom-up propagation of probabilistic choices through the nested structure, a decision tree, of the agent program. So, while the approach of Hindriks et al. can be seen as a step towards look-ahead like reactive planning, *P-BSM* remains a purely reactive approach to programming cognitive agents. Informally, except for the nested structuring of agent programs (the distinguishing practical feature of the *BSM* framework w.r.t. to other theoretically founded agent programming languages), the *P-BSM* framework could be emulated by the approach of Hindriks et al. with the look-ahead planning bound of the length one.

6 Conclusion

The main contribution of the presented paper is introduction of *Probabilistic Behavioural State Machines* framework with the associated agent programming language *Jazzyk(P)*. The proposed extension of the plain *BSM* framework is a result of practical experience with *BSM* case-studies [6] and introduces a straightforward and pragmatic, yet quite a powerful, extension of the *BSM* framework. However, the presented paper presents only first steps towards a more rigorous approach to dealing with underspecification in agent oriented programming by means of probabilistic action selection.

Underspecification of agent programs is in general inevitable. However, in situations when a suboptimal performance is tolerable, providing the agent program interpreter with a heuristics for steering its choices can lead to rapid development of more efficient and robust agent systems.

References

1. Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni. *Multi-Agent Programming Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Kluwer Academic Publishers, 2005.
2. Rafael H. Bordini, Jomi F. Hübner, and Renata Vieira. *Jason and the Golden Fleece of Agent-Oriented Programming*, chapter 1, pages 3–37. Volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations* [1], 2005.
3. Mehdi Dastani, M. Birna van Riemsdijk, and John-Jules Meyer. *Programming Multi-Agent Systems in 3APL*, chapter 2, pages 39–68. Volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations* [1], 2005.
4. Frank S. de Boer, Koen V. Hindriks, Wiebe van der Hoek, and John-Jules Ch. Meyer. A verification framework for agent programming with declarative goals. *J. Applied Logic*, 5(2):277–302, 2007.
5. Koen V. Hindriks, Catholijn M. Jonker, and Wouter Pasman. Exploring heuristic action selection in agent programming. In Koen Hindriks, Alexander Pokahr, and Sebastian Sardina, editors, *Proceedings of the Sixth International Workshop on Programming Multi-Agent Systems, ProMAS'08, Estoril, Portugal*, volume 5442 of *LNAI*, 2008.
6. Michael Köster, Peter Novák, David Mainzer, and Bernd Fuhrmann. Two case studies for Jazzyk BSM. In *Proceedings of Agents for Games and Simulations, AGS 2009, AAMAS 2009 collocated workshop, to appear*, 2009.
7. Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.
8. Andrey Andreyevich Markov. Extension of the law of large numbers to dependent quantities (in Russian). *Izvestiya Fiziko-matematicheskogo obschestva pri Kazanskom Universitete*, (15):135–156, 1906.
9. S. P. Meyn and R. L. Tweedie. *Markov Chains and Stochastic Stability*. Springer-Verlag, London, 1993.
10. Peter Novák. Jazzyk: A programming language for hybrid agents with heterogeneous knowledge representations. In *Proceedings of the Sixth International Workshop on Programming Multi-Agent Systems, ProMAS'08*, volume 5442 of *LNAI*, pages 72–87, May 2008.
11. Peter Novák and Wojciech Jamroga. Code patterns for agent-oriented programming. In *Proceedings of The Eighth International Conference on Autonomous Agents and Multi-Agent Systems, AAMAS 2009, to appear*, 2009.
12. Peter Novák and Michael Köster. Designing goal-oriented reactive behaviours. In *Proceedings of the 6th International Cognitive Robotics Workshop, CogRob 2008, July 21-22 in Patras, Greece*, pages 24–31, July 2008.
13. Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
14. Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. *Jadex: A BDI Reasoning Engine*, chapter 6, pages 149–174. Volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations* [1], 2005.
15. Anand S. Rao and Michael P. Georgeff. An Abstract Architecture for Rational Agents. In *KR*, pages 439–449, 1992.
16. Michael Winikoff. *JACK^{TM} Intelligent Agents: An Industrial Strength Platform*, chapter 7, pages 175–193. Volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations* [1], 2005.

A Proofs

Proof (Proof of Lemma 1). The proof follows by induction on nesting depth of mst's. The nesting depth of an mst is the maximal number of steps required to derive $yields_p(\tau, \sigma, p{:}\rho)$ in the $yields_p$ calculus for all σ from $\mathcal{A}_p$ and all $p{:}\rho$ yielded by τ.

$depth = 1$**:** Equation 1 is trivially satisfied for primitive updates from $\mathcal{A}_p$ of the form **skip** and $\oslash\psi$.

$depth = 2$**:** let's assume $\tau_1, \ldots, \tau_k$ are primitive mst's yielding $1{:}\rho_1, \ldots, 1{:}\rho_k$ in a state σ respectively, and ϕ be a query formula. We recognise three cases:

conditional in the case $\sigma \models \phi$, we have $yields_p(\phi \longrightarrow \tau_1, \sigma, 1{:}\rho_1)$. Similarly for $\sigma \not\models \phi$, we have $yields_p(\phi \longrightarrow \tau_1, \sigma, 1{:}\mathbf{skip})$, hence Equation 1 is satisfied in both cases.

choice according to Definition 7 for each $1 \leq i \leq k$ we have

$$yields_p(\tau_1|\cdots|\tau_k, \sigma, P_{\tau_1|\cdots|\tau_k}(\tau_i){:}\rho_i)$$

where $\Pi(\tau_1|\cdots|\tau_k) = P_{\tau_1|\cdots|\tau_k}$. Since $P_{\tau_1|\cdots|\tau_k}$ is a discrete probability distribution (cf. Definition 6) over the elements $\tau_1, \ldots, \tau_k$, we have

$$\sum_{1 \leq i \leq k} P_{\tau_1|\cdots|\tau_k}(\tau_i) = 1$$

hence Equation 1 is satisfied as well.

sequence for the sequence mst, we have $yields_p(\tau_1 \circ \cdots \circ \tau_k, \sigma, 1{:}(\rho_1 \bullet \cdots \bullet \rho_k))$, so Equation 1 is trivially satisfied again.

$depth = n$**:** assume Equation 1 holds for mst's of nesting depth $n-1$, we show it holds also for mst's of depth n. Again we assume that ϕ is a query formula of $\mathcal{A}_p$ and $\tau_1, \ldots, \tau_k$, are compound mst's of maximal nesting depth $n-1$ yielding sets of updates $\mathfrak{fp}_{\tau_1}(\sigma), \ldots, \mathfrak{fp}_{\tau_k}(\sigma)$ in a mental state σ respectively. Similarly to the previous step, we recognise three cases:

conditional according to the derivability of ϕ w.r.t. σ, for the conditional mst $\phi \longrightarrow \tau_1$ we have either $\mathfrak{fp}_{\phi\longrightarrow\tau_1}(\sigma) = \mathfrak{fp}_{\tau_1}(\sigma)$, or $\mathfrak{fp}_{\phi\longrightarrow\tau_1}(\sigma) = \{1{:}\mathbf{skip}\}$. For the latter case, Equation 1 is trivially satisfied and since τ_1 is of maximal nesting depth $n-1$, we have $\sum_{p:\rho\in\mathfrak{fp}_{\phi\longrightarrow\tau_1}(\sigma)} p = \sum_{p:\rho\in\mathfrak{fp}_{\tau_1}(\sigma)} p = 1$ as well.

choice let $P_{\tau_1|\cdots|\tau_k}$ be the probability distribution function assigned to the choice mst $\tau_1|\cdots|\tau_k$ by the function Π. We have

$$\mathfrak{fp}_{\tau_1|\cdots|\tau_k}(\sigma) = \left\{p{:}\rho | \exists 0 \leq i \leq k : yields_p(\tau_i, \sigma, p_i{:}\rho) \wedge p = P_{\tau_1|\cdots|\tau_k}(\tau_i) \cdot p_i\right\}$$

Subsequently,

$$\sum_{p:\rho\in\mathfrak{fp}_{\tau_1|\cdots|\tau_k}(\sigma)} p = \sum_{0\leq i\leq k} \left(P_{\tau_1|\cdots|\tau_k}(\tau_i) \cdot \sum_{p:\rho\in\mathfrak{fp}_{\tau_i}(\sigma)} p \right)$$

However, because of the induction assumption that Equation 1 holds for mst's τ_i with maximal nesting depth $n-1$, for all i $\sum_{p:\rho\in\mathfrak{fp}_{\tau_i}(\sigma)} p = 1$, and since $P_{\tau_1|\cdots|\tau_k}$ is a discrete probability distribution function, we finally arrive to

$$\sum_{p:\rho\in\mathfrak{fp}_{\tau_1|\cdots|\tau_k}(\sigma)} p = \sum_{0\leq i\leq k} P_{\tau_1|\cdots|\tau_k}(\tau_i) = 1$$

sequence for the sequence mst $\tau_1 \circ \cdots \circ \tau_k$, we have

$$\mathfrak{fp}_{\tau_1\circ\cdots\circ\tau_k}(\sigma) = \left\{ \prod_{i=1}^{k} p_i{:}(\rho_1 \bullet \cdots \bullet \rho_k) | \forall 1 \leq i \leq k : \mathit{yields}_p(\tau_i, \sigma, p_i{:}\rho_i) \right\}$$

and subsequently

$$\sum_{p:\rho\in\mathfrak{fp}_{\tau_1\circ\cdots\circ\tau_k}(\sigma)} p = \sum_{\prod_{i=1}^{k} p_i:(\rho_1\bullet\cdots\bullet\rho_k)\in\mathfrak{fp}_{\tau_1\circ\cdots\circ\tau_k}(\sigma)} \prod_{i=1}^{k} p_i \tag{2}$$

Observe, that if we fix the update sequence suffix $\rho_2 \bullet \cdots \bullet \rho_k$, the sum 2 can be rewritten as

$$\left(\sum_{p_1:\rho_1\in\mathfrak{fp}_{\tau_1}(\sigma)} p_1 \right) \cdot \left(\sum_{\prod_{i=2}^{k} p_i:(\rho_2\bullet\cdots\bullet\rho_k)\in\mathfrak{fp}_{\tau_2\circ\cdots\circ\tau_k}(\sigma)} \prod_{i=2}^{k} p_i \right)$$

Finally, by reformulation of the sum of products 2 as a product of sums and by applying the induction assumption for the mst's $\tau_1, \ldots, \tau_k$ of nesting depth $n-1$, we arrive to

$$\prod_{i=1}^{k} \sum_{p:\rho\in\mathfrak{fp}_{\tau_i}(\sigma)} p = \prod_{i=1}^{k} 1 = 1$$

Hence, Equation 1 is satisfied. ☐

Golog Speaks the BDI Language

Sebastian Sardina[1] and Yves Lespérance[2]

[1] RMIT University, Melbourne, Australia
[2] York University, Toronto, Canada

Abstract. In this paper, we relate two of the most well developed approaches to agent-oriented programming, namely, BDI (Belief-Desire-Intention) style programming and "Golog-like" high-level programming. In particular, we show how "Golog-like" programming languages can be used to develop BDI-style agent systems. The contribution of this paper is twofold. First, it demonstrates how *practical* agent systems can be developed using high-level languages like Golog or IndiGolog. Second, it provides BDI languages a clear classical-logic-based semantics and a powerful logical foundation for incorporating new reasoning capabilities not present in typical BDI systems.

1 Introduction

BDI (Belief-Desire-Intention) agent programming languages and platforms (e.g., PRS [11], AgentSpeak and Jason [20, 2], Jack [4], and JAM [14]) and the situation calculus-based Golog high-level programming language and its successors (e.g., ConGolog [6], IndiGolog [7, 24], and FLUX [26]) are two of the most well developed approaches within the agent-oriented programming paradigm. In this paper, we analyze the relationship between these two families of languages and show that BDI agent programming languages are closely related to IndiGolog, a situation calculus based programming language where programs are executed on-line in a dynamic environment, supporting sensing actions to acquire information from the environment and exogenous events.

BDI agent programming languages were conceived as a simplified and operationalized version of the BDI (Belief, Desire, Intention) model of agency, which is rooted in philosophical work such as Bratman's [3] theory of practical reasoning and Dennet's theory of intentional systems [8]. Practical work in the area has sought to develop programming languages that incorporate a simplified BDI semantics basis that has a computational interpretation. An important feature of BDI-style programming languages and platforms is their interleaved account of sensing, deliberation, and execution [19]. By executing as they reason, BDI agents reduce the likelihood that decisions will be made on the basis of outdated beliefs and remain responsive to the environment by making adjustments in the steps chosen as they proceed. Because of this, BDI agent programming languages are well suited to implementing systems that need to operate in "soft" real-time scenarios [16, 1]. Unlike in classical planning-based architectures, *execution* happens at each step. The assumption is that the careful crafting of plans' preconditions to ensure the selection of appropriate plans at execution time, together with a built-in mechanism for retrying alternative options, will usually ensure that a successful execution is found, even in the context of a changing environment.

In contrast to this, high-level programming languages in the Golog line aim for a middle ground between classical planning and normal programming. The idea is that the programmer may write a *sketchy* non-deterministic program involving domain specific actions and test conditions and that the interpreter will reason about these and search for a valid execution. The semantics of these languages is defined on top of the *situation calculus*, a popular predicate logic framework for reasoning about action [23]. The interpreter for the language uses an action theory representing the agent's beliefs about the state of the environment and the preconditions and effects of the actions to find a provably correct execution of the program. By controlling the amount of nondeterminism in the program, the high-level program execution task can be made as hard as classical planning or as easy as deterministic program execution. In IndiGolog, this framework is generalized to allow the programmer to control planning/lookahead and support on-line execution and sensing the environment.

In this paper, we show how a BDI agent can be built within the IndiGolog situation calculus-based programming framework. More concretely, we describe how to translate an agent programmed in a typical BDI programming language into a high-level IndiGolog program with an associated situation calculus action theory, such that *(i)* their ultimate behavior coincide; and *(ii)* the original structure of the propositional attitudes (beliefs, intentions, goals, etc.) of the BDI agent and the model of execution are preserved in the IndiGolog translation. We first do this for what we call the *core* engine of BDI systems, namely, the reactive context-sensitive expansion of events/goals. After this, we show how to accommodate more sophisticated BDI reasoning mechanisms such as goal failure recovery. In doing so, we highlight the potential additional advantages of programming BDI agents in the situation calculus, by pointing out different reasoning about action techniques that IndiGolog BDI agents may readily incorporate.

2 Preliminaries

2.1 BDI Programming

BDI agent systems were developed as a way of enabling *abstract plans* written by programmers to be combined and used in real-time, in a way that is both flexible and robust. A BDI system responds to *events*, the inputs to the system, by selecting a plan from the *plan library*, and placing it into the *intention base*, thus committing to the plan for responding to the event/goal in question. The execution of this plan-strategy may, in turn, post new subgoal events to be achieved. The plan library stands for a collection of pre-defined *hierarchical plans* indexed by goals (i.e., events) and representing the standard operations of the domain. There are a number of agent programming languages and development platforms in the BDI tradition, such as PRS [11], AgentSpeak and Jason [20, 2], Jack [4], SPARK [17], Jack [4], and JADEX [18]. Our discussion is based on the CAN family of BDI languages [27, 25], which are AgentSpeak-like languages with a semantics capturing the common essence of typical BDI systems.

A BDI agent Υ is a configuration tuple $\langle \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle$, where $\mathcal{B}$ stands for the agent's current beliefs about the world, generally a set of atoms, Π is the (static) plan-library, $\mathcal{A}$ is the sequence of actions executed so far, and Γ is the multi-set of intentions the agent is currently pursuing. The *plan library* contains plan rules of the form $e : \psi \leftarrow P$, where e

is an event/goal that triggers the plan, ψ is the context for which the plan may be applied (i.e., the precondition of the rule), and P is the body of the plan rule—P *is a reasonable strategy in order to resolve the event/goal* e *when condition* ψ *is believed to hold*. The *plan-body* P is a program built from primitive actions act that the agent can execute directly (e.g., $drive(loc1, loc3)$), operations to add $+b$ and delete $-b$ beliefs, tests for conditions $?\phi$, and (internal) subgoaling event posting $!e$ (e.g., $!Travel(mel, yyz)$). Complex plan bodies are built with the usual sequence ; and concurrency $\parallel$ constructs. There are also a number of auxiliary constructs internally used when assigning semantics to programs: the empty (terminating) program *nil*; the construct $P_1 \rhd P_2$, which tries to execute P_1, falling back to P_2 if P_1 is not possible; and $(\!|\psi_1 : P_1, \ldots, \psi_n : P_n|\!)$, which encodes a set of guarded plans. Lastly, the *intention base* Γ contains the current, partially instantiated, plan-body programs that the agent has already committed to for handling some events—since Γ is a multi-set it may contain a program more than once.

As with most BDI agent programming languages, the Plotkin-style operational semantics of CAN closely follows Rao and Georgeff's abstract interpreter for intelligent rational agents [22]: *(i)* incorporate any pending external events; *(ii)* select an intention and execute a step; and *(iii)* update the set of goals and intentions. A *transition relation* $C \longrightarrow C'$, on so-called *configurations* is defined by a set of *derivation rules* and specifies that executing configuration C a *single step* yields configuration C'. A *derivation rule* consists of a, possibly empty, set of premises, typically involving the existence of transitions together with some auxiliary conditions, and a single transition conclusion derivable from these premises. Two transition systems are used to define the semantics of the CAN language. The first transition relation $\longrightarrow$ defines what it means to execute a single intention and is defined in terms of *intention-level* configurations of the form $\langle \Pi, \mathcal{B}, \mathcal{A}, P \rangle$ consisting of the agent's plan-library Π and belief base $\mathcal{B}$, the actions $\mathcal{A}$ executed so far, and the program P being executed. The second transition relation $\Longrightarrow$ is defined in terms of the first and characterizes what it means to execute a whole agent.

So, the following are some of the intention-level derivation rules for the language:[3]

$$\frac{\Delta = \{\psi : P \mid\ e : \psi \leftarrow P \in \Pi\}}{\langle \Pi, \mathcal{B}, \mathcal{A}, !e \rangle \longrightarrow \langle \Pi, \mathcal{B}, \mathcal{A}, (\!|\Delta|\!) \rangle} Ev \quad \frac{\langle \Pi, \mathcal{B}, \mathcal{A}, P_1 \rangle \longrightarrow \langle \Pi, \mathcal{B}', \mathcal{A}', P_1' \rangle}{\langle \Pi, \mathcal{B}, \mathcal{A}, P_1 \rhd P_2 \rangle \longrightarrow \langle \Pi, \mathcal{B}', \mathcal{A}', P_1' \rhd P_2 \rangle} \rhd$$

$$\frac{\mathcal{B} \models \phi\theta}{\langle \mathcal{B}, \mathcal{A}, ?\phi \rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, nil \rangle} ? \quad \frac{\psi : P \in \Delta \quad \mathcal{B} \models \psi\theta}{\langle \Pi, \mathcal{B}, \mathcal{A}, (\!|\Delta|\!) \rangle \longrightarrow \langle \Pi, \mathcal{B}', \mathcal{A}', P\theta \rhd (\!|\Delta \setminus \{\psi : P\}|\!) \rangle} Sel$$

Derivation rule Ev captures the first stage in the plan selection process for a (pending) event-goal e, in which the agent collects, from the plan library, the set $(\!|\Delta|\!)$ of the so-called "*relevant*" (guarded) plans that may be used to resolve the pending event. Such set is later used by rules Sel and $\rhd$ to commit to and execute, respectively, an *applicable* strategy/plan P (one whose condition ψ is believed true). Notice in rule Sel how the remaining non-selected plans are kept as backup plans as the second program in the $\rhd$ construct. Finally, rule $?$ accounts for transitions over a basic test program.

On top of these intention-level derivation rules, the set of agent-level derivation rules are defined. Basically, an agent transition involves either assimilating external events

[3] Configurations must also include a variable substitution θ for keeping track of all bindings done so far during the execution of a plan-body. For legibility, we keep substitutions implicit in places where they need to be carried across multiple rules (e.g., in rule ?).

from the environment or executing an active intention. Also, in the rules below, the following auxiliary function is used to represent the set of achievement events caused by belief changes: $\Omega(\mathcal{B}, \mathcal{B}') = \{!b^- \mid \mathcal{B} \models b,\ \mathcal{B}' \not\models b\} \cup \{!b^+ \mid \mathcal{B} \not\models b,\ \mathcal{B}' \models b\}$.

$$\frac{\mathcal{E} \text{ is a set of external events} \quad \mathcal{B}' = (\mathcal{B} \setminus \{b \mid -b \in \mathcal{E}\}) \cup \{b \mid +b \in \mathcal{E}\}}{\langle \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle \Longrightarrow \langle \Pi, \mathcal{B}', \mathcal{A}, \Gamma \uplus \{!e \mid !e \in \mathcal{E}\} \uplus \Omega(\mathcal{B}, \mathcal{B}') \rangle} \; A_{ext}$$

$$\frac{P \in \Gamma \quad \langle \Pi, \mathcal{B}, \mathcal{A}, P \rangle \longrightarrow \langle \Pi, \mathcal{B}', \mathcal{A}', P' \rangle}{\langle \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle \Longrightarrow \langle \Pi, \mathcal{B}', \mathcal{A}', (\Gamma \setminus \{P\}) \uplus \{P'\} \uplus \Omega(\mathcal{B}, \mathcal{B}') \rangle} \; A_{exec}$$

Rule A_{ext} assimilates a set of external events, both achievement ones, of the form $!e$, as well as belief updates events, of the form $+b$ or $-b$—both the belief and intention bases of the agent may be updated. Note that, by means of auxiliary function Ω, a new (achievement) event of the form $!b^+$ or $!b^-$ is posted for each belief b that changes due to an external belief update; such an event may in turn trigger some new behavior.

Rule A_{exec} states that the agent may evolve one step if an active intention P can be advanced one step with remaining intention P' being left to execute. In such a case, the intention base is updated by replacing P with P' and including the belief update events produced by potential changes in the belief base. Observe that the intention base is a *multi-set*, which means that it may contain several occurrences of the same intention.

Relative to the above derivation rules, one can formally define the meaning of an agent as its possible execution traces. (See [27, 25] for the complete semantics.)

Definition 1 (BDI Execution). *A BDI execution E of an agent $\Upsilon_0 = \langle \Pi, \mathcal{B}_0, \mathcal{A}_0, \Gamma_0 \rangle$, relative to an environment $\mathcal{E}$, is a, possibly infinite, sequence of agent configurations $\Upsilon_0 \cdot \Upsilon_1 \cdot \ldots \cdot \Upsilon_n \cdot \ldots$ such that $C_i \Longrightarrow C_{i+1}$, for all $i \geq 0$. A terminating execution is a finite execution $\Upsilon_0 \cdot \ldots \cdot \Upsilon_n$ where $\Upsilon_n = \langle \Pi, \mathcal{B}_n, \mathcal{A}_n, \{\} \rangle$.*

2.2 High-Level Programming in Golog

The *situation calculus* [23] is a logical language specifically designed for representing dynamically changing worlds in which all changes are the result of named *actions*. The constant S_0 is used to denote the initial situation where no actions have yet been performed. Sequences of actions are built using the function symbol do: $do(a, s)$ denotes the successor situation resulting from performing action a in situation s. Relations whose truth values vary from situation to situation are called *fluents*, and are denoted by predicate symbols taking a situation term as their last argument (e.g., *Holding*(x, s)). A special predicate *Poss*(a, s) is used to state that action a is executable in s.

Within this language, we can formulate action theories describing how the world changes as the result of the available actions. For example, a *basic action theory* [23] includes domain-independent foundational axioms to describe the structure of situations, one successor state axiom per fluent (capturing the effects and non-effects of actions), one precondition axiom per action, and initial state axioms describing what is true initially (i.e., what is true in the initial situation S_0).

On top of situation calculus action theories, logic-based programming languages can be defined, which, in addition to the primitive actions, allow the definition of complex actions. Golog [15], the first situation calculus agent language, provides all the

usual control structures (e.g., sequence, iteration, conditional, etc.) plus some nondeterministic constructs allowing the programmer to write "sketchy" plans. ConGolog [6] extends Golog to support concurrency. To provide an intuitive overview of the language, consider the following nondeterministic program for an agent that goes to work in the morning (shamelessly taken from Ryan Kelly):

proc $goToWork$
$\quad$ *ringAlarm*; (*hitSnooze*; *ringAlarm*)*; *turnOffAlarm*;
$\quad$ $(\pi food)$[*Edible*$(food)$?; *eat*$(food)$];
$\quad$ (*haveShower* $\|$ *brushTeeth*);
$\quad$ (*driveToUni* $|$ *trainToUni*);
$\quad$ (*Time* $< 11:00$)?
endProc

While this high-level program provides a general strategy for getting up and going to work, it is underspecified, and many details, such as what to eat and how to travel to work, are left open. Program $\delta_1 \mid \delta_2$ nondeterministically chooses between programs δ_1 and δ_2, $\pi x.\, \delta(x)$ executes program $\delta(x)$ for *some* legal binding for variable x, and δ^* performs δ zero or more times. Concurrency is supported by the following three constructs: $(\delta_1 \| \delta_2)$ expresses the concurrent execution (interpreted as interleaving) of programs δ_1 and δ_2; $\delta_1 \rangle\!\rangle \delta_2$ expresses the concurrent execution of δ_1 and δ_2 with δ_1 having higher priority; and $\delta^{\|}$ executes δ zero or more times concurrently. Note that a concurrent process may become (temporarily) blocked when it reaches a test/wait action ϕ? whose condition ϕ is false (or a primitive action whose precondition is false). Test/wait actions can also be used to control which nondeterministic branches can be executed, e.g. $[(\phi?;\delta_1) \mid (\neg\phi?;\delta_2)]$, and to constrain the value of a nondeterministically bound variable, e.g., $\pi x.[\phi(x)?;\delta(x)]$. Finally, the language also accommodates the standard if-then-elses, while loops, and recursive procedures.

Finding a legal execution of a high-level program is at the core of the whole approach. Originally, Golog and ConGolog programs were intended to be executed *offline*, that is, a complete execution was obtained before committing even to the first action. However, IndiGolog [7, 24], the latest language within the Golog family, provides a formal logic-based account of interleaved planning, sensing, and action by executing programs *online* and using a specialized new construct $\Sigma(\delta)$, the *search operator*, to perform local offline planning when required.

Roughly speaking, an *online* execution of a program finds a next possible action, executes it in the real world, then obtains sensing information, and repeats the cycle until the program is completed. Formally, an online execution is a sequence of so-called online configuration of the form (δ, σ), where δ is a high-level program and σ is a history (see [7] for its formal definition). A history contains the sequence of actions executed so far as well as the sensing information obtained. Online executions are characterized in terms of the following two predicates [6]: *Final*(δ, s) holds if program δ may legally terminate in situation s; and *Trans*(δ, s, δ', s') holds if a single step of program δ in situation s may lead to situation s' with δ' remaining to be executed. In the next section, we will generalize the notion of online execution to suit our purposes.

3 BDI Programming in IndiGolog

Programming a BDI agent in the situation calculus amounts to developing a special basic action theory and a special IndiGolog high-level agent program to be executed with it. From now on, let $\Upsilon = \langle \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle$ be the BDI agent to program in IndiGolog.

3.1 The BDI Basic Action Theory

We start by showing how to obtain an action theory $\mathcal{D}^{\Upsilon}$ for our agent Υ. We assume that Υ is stated over a first-order language $\mathcal{L}_{BDI}$ containing finitely many belief and event atomic relations, namely, $b_1(\boldsymbol{x_1}), \ldots, b_n(\boldsymbol{x_n})$ and $e_1(\boldsymbol{x_1}), \ldots, e_m(\boldsymbol{x_n})$.

Let us then define what the fluents and actions available in the situation calculus language $\mathcal{L}_{sitCalc}$ are. First, for every belief atomic predicate $b(\boldsymbol{x})$ in $\mathcal{L}_{BDI}$, the language $\mathcal{L}_{sitCalc}$ includes a relational fluent $b(\boldsymbol{x}, s)$ together with two primitive actions $add_b(\boldsymbol{x})$ and $del_b(\boldsymbol{x})$ which are meant to change the fluent's truth value. Second, for each achievement event type $e(\boldsymbol{x})$ in the domain, there is a corresponding action term $ach_e(\boldsymbol{x})$ in $\mathcal{L}_{sitCalc}$. Finally, for every action atom $A(\boldsymbol{x})$ in $\mathcal{L}_{BDI}$, there is a corresponding action function $A(\boldsymbol{x})$ in $\mathcal{L}_{sitCalc}$.

In addition, the language $\mathcal{L}_{sitCalc}$ shall include one auxiliary distinguished fluent and two actions to model external event handling. Fluent $\mathit{PendingEv}(s)$ stands for the multi-set of events that are "pending" and need to be handled, either belief update or achievement events. This fluent is affected by two actions. Whereas action $\mathit{post}(e)$ indicates the external posting of event e; action $\mathit{serve}(e)$ indicates that (pending) event e has been selected and is being handled. In both actions, argument e is of sort action.

Let us now construct the basic action theory $\mathcal{D}^{\Upsilon}$ corresponding to a BDI agent $\Upsilon = \langle \Pi, \mathcal{B}, \Gamma \rangle$, as follows:

1. The initial description in $\mathcal{D}^{\Upsilon}$ is defined in the following way:

$$\mathcal{D}^{\Upsilon}_{S_0} = \bigcup_{i=1}^{n} \{\forall \boldsymbol{x}. b_i(\boldsymbol{x}, S_0) \equiv \boldsymbol{x} = \boldsymbol{t_i^1} \vee \ldots \vee \boldsymbol{x} = \boldsymbol{t_i^{k_i}}\} \,\cup \\ \{\forall a. \mathit{Exog}(a) \equiv (\exists a') a = \mathit{post}(a')\},$$

   where for every $i \in \{1, \ldots, n\}$, $\mathcal{B} \models b_i(\boldsymbol{x}) \equiv [\boldsymbol{x} = \boldsymbol{t_i^1} \vee \ldots \vee \boldsymbol{x} = \boldsymbol{t_i^{k_i}}]$, for some $k_i \geq 0$—$b_i(\boldsymbol{t_i^1}), \ldots, b_i(\boldsymbol{t_i^{k_i}})$ are all the true belief atoms in $\mathcal{B}$ with respect to belief relation b_i (each $\boldsymbol{t_i^j}$ is a vector of ground terms).
2. The following precondition axioms, for every fluent $b(\boldsymbol{x})$ and action $A(\boldsymbol{x})$:

$$\mathit{Poss}(\mathit{serve}(a), s) \equiv (a \in \mathit{PendingEv}(s)) \qquad \mathit{Poss}(A(\boldsymbol{x}), s) \equiv \mathtt{True}$$
$$\mathit{Poss}(\mathit{add}_b(\boldsymbol{x}), s) \equiv \mathit{Poss}(\mathit{del}_b(\boldsymbol{x}), s) \equiv \mathtt{True} \qquad \mathit{Poss}(\mathit{post}(a), s) \equiv \mathtt{True}$$

3. For every domain fluent $b(\boldsymbol{x}, s)$, $\mathcal{D}^{\Upsilon}$ includes the following successor state axiom:

$$b(\boldsymbol{x}, do(a, s)) \equiv \\ a = \mathit{add}_b(\boldsymbol{x}) \vee a = \mathit{post}(\mathit{add}_b(\boldsymbol{x})) \vee b(\boldsymbol{x}, s) \wedge (a \neq \mathit{del}_b(\boldsymbol{x}) \wedge a \neq \mathit{post}(\mathit{del}_b(\boldsymbol{x})).$$

   That is, the truth-value of fluent b is affected only by actions add_b and del_b, either internally executed or externally sensed from the environment.

More importantly, action theory $\mathcal{D}^{\Upsilon}$ includes a successor state axiom for fluent *PendingEv*$(do(a,s))$ specifying how the multi-set of pending events changes:

$$\mathit{PendingEv}(do(a,s)) = v \equiv [\gamma(a,v,s) \vee \mathit{PendingEv}(s) = v \wedge \neg\exists v'.\gamma(a,v',s)];$$

where:

$$\gamma(a,v,s) \stackrel{\text{def}}{=} \big(\bigvee_{i=1}^{n}[\gamma_i^{+}(a,v,s) \vee \gamma_i^{-}(a,v,s)] \vee \bigvee_{i=1}^{m}[\gamma_i^{e}(a,v,s)] \vee \\ \exists a'.a = \mathit{serve}(a') \wedge v = \mathit{PendingEv}(s) \setminus \{a'\}\big);$$

$$\gamma_i^{+}(a,v,s) \stackrel{\text{def}}{=} \\ \exists \boldsymbol{x}.\, a \in \{\mathit{add}_{b_i}(\boldsymbol{x}), \mathit{post}(\mathit{add}_{b_i}(\boldsymbol{x}))\} \wedge \neg b_i(\boldsymbol{x}) \wedge v = \mathit{PendingEv}(s) \uplus \{\mathit{add}_{b_i}(\boldsymbol{x})\};$$

$$\gamma_i^{-}(a,v,s) \stackrel{\text{def}}{=} \\ \exists \boldsymbol{x}.\, a \in \{\mathit{del}_{b_i}(\boldsymbol{x}), \mathit{post}(\mathit{del}_{b_i}(\boldsymbol{x}))\} \wedge b_i(\boldsymbol{x}) \wedge v = \mathit{PendingEv}(s) \uplus \{\mathit{del}_{b_i}(\boldsymbol{x})\};$$

$$\gamma_i^{e}(a,v,s) \stackrel{\text{def}}{=} \exists \boldsymbol{x}.\, a = \mathit{post}(\mathit{ach}_{e_i}(\boldsymbol{x})) \wedge v = \mathit{PendingEv}(s) \uplus \{\mathit{ach}_{e_i}(\boldsymbol{x})\}.$$

That is, an actual change in the belief of an atom, either due to the execution of some intention or an external event, automatically produces a corresponding pending belief update event. Moreover, an external achievement event $\mathit{ach}_e(\boldsymbol{x})$ becomes pending when sensed. On the other hand, an event e ceases to be pending when action *serve*(e) is executed.

4. Theory $\mathcal{D}^{\Upsilon}$ includes unique name axioms for all actions in $\mathcal{L}_{sitCalc}$, as well as the standard domain-independent foundational axioms for the situation calculus ([23]).

This concludes the construction of the BDI basic action theory $\mathcal{D}^{\Upsilon}$.

3.2 The BDI Agent Program

Let us now construct the IndiGolog BDI agent program δ^{Υ} that is meant to execute relative to the BDI action theory $\mathcal{D}^{\Upsilon}$. We start by showing how to inductively transform a BDI plan-body program P into an IndiGolog program δ_P, namely (remember that plan-bodies programs are used to build BDI plans in the plan library):

$$\delta_P = \begin{cases} P & \text{if } P = \mathit{act} \mid \mathit{nil} \\ \phi? & \text{if } P = ?\phi \\ \mathit{add}_b(\boldsymbol{t}) & \text{if } P = +b(\boldsymbol{t}) \\ \mathit{del}_b(\boldsymbol{t}) & \text{if } P = -b(\boldsymbol{t}) \\ \mathit{handle}(\mathit{ach}_e(\boldsymbol{t})) & \text{if } P = !e(\boldsymbol{t}) \\ (\delta_{P_1};\delta_{P_2}) & \text{if } P = (P_1;P_2) \\ \delta_{P_1} & \text{if } P = P_1 \rhd P_2 \\ \mathit{achieve}_e(\boldsymbol{t}) & \text{if } P = (\!|\Delta|\!), \text{for some event } e(\boldsymbol{t}) \\ (\delta_{P_1};\delta_{P_2}) & \text{if } P = (P_1;P_2) \\ (\delta_{P_1};\delta_{P_2}) & \text{if } P = (P_1;P_2) \end{cases}$$

Notice that achievement events $!e$ occurring in a plan are handled via simple plan invocation, by invoking procedure *handle*; a new top-level intention is *not* created.

Next, we describe how to transform the BDI plans in the agent's plan library. To that end, suppose that $e(\boldsymbol{x})$ is an event in the BDI language $\mathcal{L}_{BDI}$ such with the following $n \geq 0$ plans in Π ($\boldsymbol{v_t}$ denotes all the distinct free variables in the terms $\boldsymbol{t}$):

$$e(\boldsymbol{t_i}) : \psi_i(\boldsymbol{v_{t_i}}, \boldsymbol{y_i}) \leftarrow P_i(\boldsymbol{v_{t_i}}, \boldsymbol{y_i}, \boldsymbol{z_i}),\ \text{where } i \in \{1,\ldots,n\}.$$

Then, we build the following high-level Golog procedure with n non-deterministic choices (i.e., as many as plan-rules for the event):

$$\begin{array}{l}\textbf{proc}\ \mathit{achieve}_e(\boldsymbol{x})\\ \quad |_{i\in\{1,\ldots,n\}}\ [(\pi \boldsymbol{v_{t_i}}, \boldsymbol{y_i}, \boldsymbol{z_i}).(\boldsymbol{x} = \boldsymbol{t_i} \wedge \psi_i(\boldsymbol{v_{t_i}}, \boldsymbol{y_i}))?; \delta_{P_i}(\boldsymbol{v_{t_i}}, \boldsymbol{y_i}, \boldsymbol{z_i})]\\ \textbf{endProc}\end{array}$$

Roughly speaking, executing $\mathit{achieve}_e(\boldsymbol{x})$ involves nondeterministically choosing among the n available options in the plan library for event e. See that the first test statement in each option amounts to checking the relevance and applicability of the option. Thus, the execution of $\mathit{achieve}_e(\boldsymbol{x})$ is bound to *block* if no option is relevant or applicable. In particular, the procedure will *always* block if the agent Υ has no plan to handle the event in question—that is, if $n = 0$, the corresponding Golog procedure is simply $?(\texttt{False})$.

Let Δ_Π denote the set of Golog procedures as above, one per event in the BDI language, together with the following procedure:

$$\begin{array}{l}\textbf{proc}\ \mathit{handle}(a)\\ \quad |_{i=1}^{n}\ [(\exists \boldsymbol{x_i}.a = \mathit{add}_{b_i}(\boldsymbol{x_i}))?;\ \mathit{achieve}_{b_i^+}(\boldsymbol{x_i})]\ |\\ \quad |_{i=1}^{n}\ [(\exists \boldsymbol{x_i}.a = \mathit{del}_{b_i}(\boldsymbol{x_i}))?;\ \mathit{achieve}_{b_i^-}(\boldsymbol{x_i})]\ |\\ \quad |_{i=1}^{m}\ [(\exists \boldsymbol{x_i}.a = \mathit{ach}_{e_i}(\boldsymbol{x_i}))?;\ \mathit{achieve}_{e_i}(\boldsymbol{x_i})]\\ \textbf{endProc}\end{array}$$

That is, when a is a legal event (belief update or achievement goal), procedure $\mathit{handle}(a)$ calls the appropriate procedure that is meant to *resolve* the event. Observe that this program contains two nondeterministic programs per belief atom in the domain (one to handle its addition and one to handle its deletion from the belief base), plus one nondeterministic program per achievement event in the domain.

Finally, we define the top-level IndiGolog BDI agent program as follows:

$$\delta^\Upsilon \stackrel{\text{def}}{=} \Delta_\Pi; [\delta_{env} \| \ (\delta_\Gamma \parallel \delta_{BDI})]; (\neg\exists e\ \mathit{PendingEv}(e))?, \tag{1}$$

where (assuming that $\Gamma = \{P_1, \ldots, P_n\}$):

$$\delta_\Gamma \stackrel{\text{def}}{=} \delta_{P_1} \| \cdots \| \ \delta_{P_n}; \quad \delta_{env} \stackrel{\text{def}}{=} (\pi a.\mathit{Exog}(a)?; a)^*. \quad \delta_{BDI} \stackrel{\text{def}}{=} [\pi a.\mathit{serve}(a); \mathit{handle}(a)]^{\|};$$

The set of programs Δ_Π provides the environment encoding the BDI plan library. Program δ_Γ accounts for all current intentions in Υ; if $\Gamma = \emptyset$, then $\delta_\Gamma = \mathit{nil}$. In turn, program δ_{env} models the external environment, which can perform zero, one, or more actions of the form $\mathit{post}(a)$, representing an external achievement event goal ($a = \mathit{ach}_e(\boldsymbol{t})$) or a belief update event ($a = \mathit{add}_b(\boldsymbol{t})$ or $a = \mathit{del}_b(\boldsymbol{t})$).

The most interesting part of δ^Υ is indeed the ConGolog program δ_{BDI}, which implements (part of) the BDI execution cycle. More concretely, δ_{BDI} is responsible for selecting an external event and *spawning* a new "intention" concurrent thread for handling it. To that end, δ_{BDI} picks an event a (e.g., $\mathit{add}_{At}(23, 32)$ or $\mathit{achieve}_{moveTo}(0, 0)$) to be served and executes action $\mathit{serve}(a)$. Observe that an event can be served only if it is currently pending (see action precondition for action $\mathit{serve}(a)$ in Subsection 3.1). After the action $\mathit{serve}(a)$ has been successfully executed, the selected event a is actually

handled, by calling procedure *handle*(a) defined in Δ_{Π}. More importantly, this is done in a "new" concurrent thread, so that program δ_{BDI} is still able to serve and handle other pending events. The use of concurrent iteration to spawn a new intention from the "main BDI thread" is inspired from the server example application in [6].

Note that Δ_{Π} and δ_{Γ} are domain dependent, i.e., they are built relative to a particular BDI agent Υ, whereas programs δ_{BDI} and δ_{env} are independent of the BDI agent being encoded. Observe also that the whole high-level program δ^{Υ} may terminate only when no more events are pending.

From now on, let $\mathcal{G}^{\Upsilon} = \langle \mathcal{D}^{\Upsilon}, \delta^{\Upsilon}, \mathcal{A} \rangle$ denote the IndiGolog agent for BDI agent Υ.

3.3 LC-Online Executions

Once we have a BDI IndiGolog program $\mathcal{G}^{\Upsilon}$ on hand, we should be able to execute it and obtain the same behavior and outputs as with the original BDI agent. Unfortunately, we cannot execute $\mathcal{G}^{\Upsilon}$ *online*, as defined in [7], as such executions may commit too early to free variables in a program—online executions are sequences of *ground* online configurations. What we need, instead, is an account of execution that commits to free variables only when necessary. To that end, we generalize the online execution notion from [7] to what we call *least-committed* online executions. We first define two meta-theoretic versions of relations *Trans* and *Final* as follows:

$$mTrans(\delta(\boldsymbol{x}, \boldsymbol{y}), \sigma, \delta'(\boldsymbol{x}, \boldsymbol{z}), \sigma') \stackrel{\text{def}}{=} \\ Axioms[\mathcal{D}, \sigma] \models \exists \boldsymbol{y} \forall \boldsymbol{x}, \boldsymbol{z}. Trans(\delta(\boldsymbol{x}, \boldsymbol{y}), end[\sigma], \delta'(\boldsymbol{x}, \boldsymbol{z}), end[\sigma']);$$

$$mFinal(\delta(\boldsymbol{x}, \boldsymbol{y}), \sigma) \stackrel{\text{def}}{=} Axioms[\mathcal{D}, \sigma] \models \exists \boldsymbol{x}. Final(\delta(\boldsymbol{x}), end[\sigma]).$$

(Here, in $\delta(\boldsymbol{x})$ the vector of variables $\boldsymbol{x}$ contains *all* the free variables mentioned in the program, and different variables vectors are assumed disjoint; *end*$[\sigma]$ denotes the situation term corresponding to the history σ; and *Axioms*$[\mathcal{D}, \sigma]$ denotes the complete set of axioms in the IndiGolog theory, which includes the action theory $\mathcal{D}$ for the domain and all the axioms for *Trans* and *Final*.)

We can then define least-committed executions as follows.

Definition 2 (LC-Online Execution). *A least-committed online (lc-online) execution of an IndiGolog program δ starting from a history σ is a, possibly infinite, sequence of configurations $(\delta_0 = \delta, \sigma_0 = \sigma), (\delta_1, \sigma_1), \ldots$ such that for every $i \geq 0$:*

1. *$mTrans(\delta_i, \sigma_i, \delta_{i+1}, \sigma_{i+1})$ holds; and*
2. *for all δ' such that $mTrans(\delta_i, \sigma_i, \delta', \sigma_{i+1})$ and $\delta_{i+1} = \delta'\theta$ for some substitution θ, there exists θ' such that $\delta' = \delta_{i+1}\theta'$.*

A finite lc-online execution $(\delta_0, \sigma_0), \ldots, (\delta_n, \sigma_n)$ is terminating iff $mFinal(\delta_n, \sigma_n)$ or for all δ', σ' $mTrans(\delta_n, \sigma_n, \delta', \sigma')$ does not hold.

We notice that, as expected, it can be shown that an lc-online execution stands for all its ground online instances as defined in [7]. However, by executing programs in a least committed way, we avoid premature binding of variables and eliminate some executions where the program is bound to fail.

3.4 BDI/IndiGolog Bisimulation

We are now ready to provide the main results of the paper. Namely, we show that given any BDI execution of an agent, there exists a matching execution of the corresponding IndiGolog agent, and vice-versa. In addition, the correspondence in the internal structure of the agents is always maintained throughout the executions.

We start by characterizing when a BDI agent and an IndiGolog agent configuration "match." To that end, we shall use relation $\Upsilon \approx \mathcal{G}$, which, intuitively, holds if a BDI agent Υ and an IndiGolog agent $\mathcal{G}$ represent the same (BDI) agent system. Formally, relation $\langle \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle \approx \langle \mathcal{D}, \delta, \sigma \rangle$ holds iff

1. $\delta = \Delta_\Pi; [\delta_{env} \| (\delta_{\Gamma'} \parallel \delta_{BDI})]; ?(\neg \exists e\, \mathit{PendingEv}(e))$, for some $\Gamma' \subseteq \Gamma$ such that $\Gamma = \Gamma' \uplus \{a \mid \mathit{Axioms}[\mathcal{D}, \sigma] \models a \in \mathit{PendingEv}(\mathit{end}[\sigma])\}$;
2. $\mathcal{A}$ and σ contain the same sequence of *domain* actions;
3. for every ground belief atom b: $\mathcal{B} \models b$ iff $\mathit{Axioms}[\mathcal{D}, \sigma] \models b[\mathit{end}[\sigma]]$;
4. $\mathcal{D} = \mathcal{D}^{\Upsilon'}$, for some $\Upsilon' = \langle \Pi, \mathcal{B}', \mathcal{A}, \Gamma \rangle$.

The first condition states that the IndiGolog program is of the form shown in equation (1) above (see Section 3.2), but where some active intentions may still be "pending." In other words, some intentions in Γ that have not yet started execution may not show up yet as concurrent processes in δ, but they are implicitly represented as "pending" in fluent $\mathit{PendingEv}(s)$. The second requirement states that both agents have performed the same sequence of domain primitive actions, that is, actions other than the internal ones $\mathit{serve}(a)$, $\mathit{post}(a)$, $\mathit{add}_b(\boldsymbol{x})$, and $\mathit{del}_b(\boldsymbol{x})$. The third condition requires both agents to coincide on what they believe. Finally, the IndiGolog agent executes relative to a basic action theory whose dynamics are as described in Section 3.1. Observe that the *initial* beliefs of the IndiGolog do not necessarily need to coincide with those of the BDI agent, as long as the *current* beliefs do (that is, the beliefs hold after history σ).

First of all, it is possible to show that the encoding of initial BDI agents, that is agents that have not yet performed any action, into IndiGolog agents described above is in the $\approx$ relation with the original BDI agent.

Theorem 1. *Let Υ be an initial BDI agent (that is, $\mathcal{A} = \epsilon$). Then, $\Upsilon \approx \langle \mathcal{D}^\Upsilon, \delta^\Upsilon, \mathcal{A} \rangle$.*

The importance of a BDI agent and an IndiGolog agent being in the $\approx$ relation is that their respective transitions can then always be simulated by the other type of agent To demonstrate that, we first show that any BDI transition can be replicated by the corresponding IndiGolog agent. Observe that IndiGolog may need several transitions to replicate the BDI transition when it comes to assimilating external events; whereas BDI agents incorporate sets of external events in a single transition, the IndiGolog agent incorporates one event per transition. Also, IndiGolog agents ought to execute the special action $\mathit{serve}(a)$ to start handling external achievement events.

Theorem 2. *Let Υ be a BDI agent and $\langle \mathcal{D}, \delta, \sigma \rangle$ an IndiGolog agent such that $\Upsilon \approx \langle \mathcal{D}, \delta, \sigma \rangle$. If $\Upsilon \Longrightarrow \Upsilon'$, then there exists a program δ' and a history σ' such that $mTrans^*(\delta, \sigma, \delta', \sigma')$ holds relative to action theory $\mathcal{D}$, and $\Upsilon' \approx \langle \mathcal{D}, \delta', \sigma' \rangle$.*

Furthermore, in the other direction, any step in a BDI IndiGolog execution can always be "mimicked" by the corresponding BDI agent.

Theorem 3. *Let Υ and $\langle \mathcal{D}, \delta, \sigma \rangle$ be a BDI and an IndiGolog agents, respectively, such that $\Upsilon \approx \langle \mathcal{D}, \delta, \sigma \rangle$. Suppose that $mTrans(\delta, \sigma, \delta', \sigma')$ holds relative to action theory $\mathcal{D}$, for some IndiGolog program δ' and history σ'. Then, either $\Upsilon \approx \langle \mathcal{D}, \delta', \sigma' \rangle$ or there exists a BDI agent Υ' such that $\Upsilon \Longrightarrow \Upsilon'$ and $\Upsilon' \approx \langle \mathcal{D}, \delta', \sigma' \rangle$.*

So, when the IndiGolog agent performs a transition it remains "equivalent" to the BDI agent or to some evolution of the BDI agent. The former case applies only when the transition in question involved the execution of a *serve*(a) action to translate a pending event into a concurrent process.

Putting both theorems together, our encoding allows IndiGolog to bisimulate BDI agents.

4 BDI Failure Handling

Since BDI systems are meant to operate in dynamic settings, plans that were supposed to work may fail due to changes in the environment. Indeed, a plan may fail because a test condition $?\phi$ is not believed true, an action cannot be executed, or a sub-goal event does not have any applicable plans. The BDI language we have discussed so far has no strategy towards failed plans or intentions, once an intention cannot evolve, it simply remains in the intention base *blocked*. In this section, we discuss how BDI programming languages typically address plan/intention failure and show how the above IndiGolog encoding can be extended accordingly. In particular, we show how agents can *abandon* failed intentions and *recover* from problematic plans by trying alternative options.

Before getting into technical details, we shall first introduce a new construct into the IndiGolog language. In "Golog-like" languages, a program that is *blocked* may not be dropped for the sake of another program. To overcome this, we introduce the construct $\delta_1 \rhd \delta_2$ with the intending meaning that δ_1 should be executed, falling back to δ_2 if δ_1 becomes blocked:[4]

$$\begin{aligned} \mathit{Trans}(\delta_1 \rhd \delta_2, s, \delta', s') \equiv\ & (\exists \gamma. \mathit{Trans}(\delta_1, s, \gamma, s') \land \delta' = \gamma \rhd \delta_2)\ \lor \\ & \neg\exists \gamma, s''. \mathit{Trans}(\delta_1, s, \gamma, s'') \land \mathit{Trans}(\delta_2, s, \delta', s'); \\ \mathit{Final}(\delta_1 \rhd \delta_2, s, \delta', s') \equiv\ & \mathit{Final}(\delta_1, s) \lor \neg\exists \gamma, s''. \mathit{Trans}(\delta_1, s, \gamma, s'') \land \mathit{Final}(\delta_2, s). \end{aligned}$$

4.1 Dropping Impossible Intentions

It is generally accepted that intentions that cannot execute further may simply be *dropped* by the agent — rational agents should not pursue intentions/goals that are deemed impossible [21, 5]. This is indeed the behavior of AgentSpeak agents.[5]

The BDI language of Section 2.1 can be easily extended to provide such an intention-dropping facility, by just adding the following agent-level operational rule:

$$\frac{P \in \Gamma \quad \langle \Pi, \mathcal{B}, \mathcal{A}, P \rangle \not\longrightarrow}{\langle \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle \Longrightarrow \langle \Pi, \mathcal{B}, \mathcal{A}, \Gamma \setminus \{P\} \rangle}\ A_{clean}$$

[4] One could easily extend these definitions to only allow dropping a blocked δ_1 under given conditions; this could be used to implement "time outs" or allow blocking for synchronization.

[5] There has been work on more sophisticated treatments of plan failure in AgentSpeak [2].

That is, an agent may choose to just drop an intention from its intention base if it cannot execute further in the current mental state. To mimic this behavior in our BDI IndiGolog formalization, we slightly modify the domain-independent program δ_{BDI} as follows:

$$\delta_{BDI} \stackrel{\text{def}}{=} [\pi a.serve(a); (handle(a) \rhd (\texttt{True})?)]^{||}.$$

Here, a pending event is handled within the scope of a $\rhd$, which basically allows the intention thread to simply terminate if it becomes blocked. Notice that, as with BDI languages, for procedure *handle*(a) to be blocked, every sub-goal event triggered by the handling of a (represented in the IndiGolog program as simple procedure calls) ought to be blocked. Observe also that in this approach, only the main program corresponding to a top-level event may be dropped, not lower-level instrumental subgoals.

4.2 BDI Goal Failure Recovery

Merely dropping a whole intention when it becomes *blocked* provides a rather weak level of commitment to goals. The failure of a plan should not be equated to the failure of its parent goal, as there could be alternative ways to achieve the latter. For example, suppose an agent has the goal to quench her thirst, and in the service of this goal, she adopts the subgoal of buying a can of soda [25]. However, upon arrival at the store, she realizes that all the cans of soda are sold out. Fortunately though, the shop has bottles of water. In this situation, it is irrational for the agent to drop the whole goal of quenching her thirst just because soda is not available. An AgentSpeak agent may do so. Similarly, we do not expect the agent to fanatically insist on her subgoal and just wait indefinitely for soda to be delivered. What we expect is the agent to merely drop her commitment to buy soda and adopt the alternative (sub)goal of buying a bottle of water, thereby achieving the *main* goal.

As a matter of fact, one of the typical features of implemented BDI languages is that of plan-goal failure recovery: if a plan happens to fail for a goal, usually due to unexpected changes in the environment, another plan is tried to achieve the goal. If no alternative strategy is available, then the goal is deemed failed and failure is propagated to higher-level motivating goals, and so on. This mechanism thus provides a stronger level of *commitment to goals*, by decoupling plan failure from goal failure. To accommodate failure handling, we further extend the BDI language of Section 2.1, by providing the following additional derivation rule for construct $\rhd$:

$$\frac{\langle \Pi, \mathcal{B}, \mathcal{A}, P_1 \rangle \not\longrightarrow \quad \langle \Pi, \mathcal{B}', \mathcal{A}', P_2' \rangle \longrightarrow \langle \Pi, \mathcal{B}', \mathcal{A}', P_2' \rangle}{\langle \Pi, \mathcal{B}, \mathcal{A}, P_1 \rhd P_2 \rangle \longrightarrow \langle \Pi, \mathcal{B}', \mathcal{A}', P_2' \rangle} \rhd_f$$

That is, if the current strategy P_1 is *blocked* but the alternative backup program P_2 is able to evolve, then it is legal to drop P_1 and switch to P_2. Observe that due to derivation rules Ev and Sel, $P_2 = (\!|\Delta|\!)$ will encode the set of *relevant* plans that have not yet been tried for the event being addressed. From now on, let the CAN language refer to our extended BDI language, with both new derivation rules A_{clean} and $\rhd_f$ for failure included.

Hence, due to the interaction between derivation rules Ev, Sel and $\rhd_f$, a CAN BDI agent executes a program $P_1 \rhd (\!|\Delta|\!)$ in order to resolve an goal event $!e$. When the

current strategy P_1 being pursued is not able to make a step, the agent may check the set of alternatives $(\!|\Delta|\!)$ in the hope of finding another option P_2 for addressing e. If one is found, the agent may opt to abandon its strategy P_1 and continue with P_2. (Details can be found in [27, 25].)

Let us now describe how to replicate this failure recovery behavior within our IndiGolog framework of Section 3. For simplicity, we shall assume that, as with actions, only *ground* posting of subgoal events are allowed in the BDI language. This means that all variables $\boldsymbol{x}$ in an event posting $!e(\boldsymbol{x})$ are considered *inputs* to the event. If an event is meant to return data, it must do so by using of the belief base. To support failure recovery, we slightly modify how plans in the plan library Π are converted into ConGolog procedures. Specifically, for each event $e(\boldsymbol{x})$, we define the following procedure (and make procedure $achieve_e(\boldsymbol{x})$ simply call $achieve'_e(\boldsymbol{x}, [1, \ldots, 1])$:

proc $achieve'_e(\boldsymbol{x}, \boldsymbol{w})$ `// ` $\boldsymbol{w}$ `is an` n`-long vector`
$|_{i \in \{1,\ldots,n\}}\ [(\pi \boldsymbol{v_{t_i}}, \boldsymbol{y_i}, \boldsymbol{z_i}).(\boldsymbol{x} = \boldsymbol{t} \wedge \psi_i(\boldsymbol{v_t}, \boldsymbol{y}) \wedge w = 1)?; \delta_{P_i}(\boldsymbol{v_{t_i}}, \boldsymbol{y_i}, \boldsymbol{z_i}) \rhd \Phi_i(\boldsymbol{x}, \boldsymbol{w})]$
endProc

where $\Phi_i(\boldsymbol{x}, \boldsymbol{w}) \stackrel{\text{def}}{=} achieve'_e(\boldsymbol{x}, [w_1, \ldots, w_{i-1}, 0, w_{i+1}, \ldots, w_n])$.

Vector $\boldsymbol{w}$ has one component per plan rule in the library for the event in question; its i-th component w_i states whether the i-th plan in Π is *available* for selection. Condition $(\boldsymbol{x} = \boldsymbol{t} \wedge \psi_i(\boldsymbol{v_t}, \boldsymbol{y}) \wedge w = 1)$ checks whether event $e(\boldsymbol{x})$ is relevant, applicable, and available. Program Φ_i determines the *recovery strategy*, in this case, recursively calling the procedure to achieve the event, but removing the current plan from consideration (by setting its component in $\boldsymbol{w}$ to 0). Due to the semantics of $\rhd$, recovery would only be triggered if procedure $achieve'_e(\boldsymbol{x}, \boldsymbol{w})$ may execute one step, which implies that there is indeed an available plan that is relevant and applicable for the event.

It turns out that these are the only modifications to the encoding of Section 3 required to mimic the behavior of CAN agents with failure handling in the IndiGolog high-level language.

Theorem 4. *Theorems 2 and 3 hold for CAN agents under the extended translation to IndiGolog agents.*

More interestingly, the proposed translation can be adapted to accommodate several alternative accounts of execution and failure-recovery. For example, goal failure recovery can be disallowed for an event by just taking $\Phi_i(\boldsymbol{x}, \boldsymbol{w}) \stackrel{\text{def}}{=} ?(\texttt{False})$ above. Similarly, a framework under which *any* plan may be (re)tried for achieving a goal event, regardless of previous (failed) executions, is obtained by taking $\Phi_i(\boldsymbol{x}, \boldsymbol{w}) \stackrel{\text{def}}{=} achieve_e(\boldsymbol{x})$. In this case, the event is “fully” re-posted within the intention.

The key point here is that, due to the fact that the BDI execution and recovery model is represented in our BDI IndiGolog at the *object* level, one can even go further and design more sophisticated accounts of execution and failure recovery for BDI agents. It is straightforward, for instance, to model the kind of goal failure recovery originally proposed for AgentSpeak, in which the system would automatically post a distinguished *failure goal* (denoted $!{-}g$); the programmer may then choose to provide plans to handle such failure events. A failure handling plan could, for example, carry out some clean-up tasks and even re-post the failed event [20, 2]. This type of behavior can be easily

achieved by taking $\Phi_i(\boldsymbol{x}, \boldsymbol{w}) \stackrel{\text{def}}{=} ach_{fail_e}(\boldsymbol{x}); ?(\texttt{False})$, and allowing the programmer to provide plan rules in the library for handling the special event $fail_e(\boldsymbol{x})$. Notice that the event is *posted* so it would eventually create a new intention all-together; the current plan would then immediately be blocked/failed.

5 Discussion

In this paper, we have shown how one can effectively program BDI-style agent systems in the situation calculus-based IndiGolog high-level programming language. The benefits of this are many. First, we gain a better understanding of the common features of BDI agent programming languages and "Golog-like" high-level programming languages, as well as of what is specific to each type of language, and what is required to reproduce BDI languages in the latter. We also get a new classical-logic situation calculus-based semantics for BDI agent programming languages. This opens many avenues for enhancing the BDI programming paradigm with reasoning capabilities, for instance, model-based belief update capabilities, lookahead planning capabilities, plan/goal achievement monitoring capabilities, etc. Our account also shows how one can essentially compile the BDI execution engine of a BDI agent into an object-level IndiGolog program, about which we can reason in the situation calculus. From the perspective of situation calculus-based high-level programming languages, we have enhanced IndiGolog with a more general semantic account of program execution, i.e. least-committed online executions, and we have also introduced a novel language construct that is useful for failure handling. Moreover, our work opens up new perspectives for developing logic-based agent programming languages with BDI features.

There has only been limited work on relating "Golog-like" and BDI programming languages. Hindriks *et al.* [13] show that ConGolog can be bisimulated by the agent language 3APL under some conditions, which include the agent having complete knowledge. In [12], it is also shown that AgentSpeak can be encoded into 3APL. Our results are complementary, in showing the inverse relationship. Note also that 3APL is a rather atypical BDI-style programming language, which for instance, does not use events. So it is interesting to have a direct comparison with classical BDI programming languages in the AgentSpeak tradition.

Also related is the work of Gabaldon [10] on encoding Hierarchical Task Network (HTN) libraries in ConGolog. There are similarities between our work and his in the way procedural knowledge is encoded in ConGolog. This is is not surprising, as HTN planning systems and BDI agents have many similarities [9]. But note that in HTNs, and hence in Gabaldon's translation, the objective is *planning* and not reactive execution. We on the other hand, focus on capturing the typical execution regime of BDI agent systems, rather than on performing lookahead planning to synthesize a solution. As a result, we address issues such as external events and plan failure that do not arise in HTN planning.

References

1. S. S. Benfield, J. Hendrickson, and D. Galanti. Making a strong business case for multiagent technology. In *Proc. of AAMAS*, pages 10–15, New York, NY, USA, 2006. ACM Press.
2. R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-agent Systems in AgentSpeak Using Jason*. Wiley, 2007. Series in Agent Technology.

3. M. E. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, 1987.
4. P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. JACK intelligent agents: Components for intelligent agents in Java*AgentLink Newsletter*, 2,1999Agent Oriented Software Pty. Ltd.
5. P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence Journal*, 42:213–261, 1990.
6. G. De Giacomo, Y. Lespérance, and H. J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Art. Intelligence Journal*, 121(1–2):109–169, 2000.
7. G. De Giacomo and H. J. Levesque. An incremental interpreter for high-level programs with sensing. In H. J. Levesque and F. Pirri, editors, *Logical Foundations for Cognitive Agents: Contributions in Honor of Ray Reiter*, pages 86–102. Springer, Berlin, 1999.
8. D. Dennett. *The Intentional Stance*. The MIT Press, 1987.
9. J. Dix, H. Muñoz-Avila, D. S. Nau, and L. Zhang. IMPACTing SHOP: Putting an AI planner into a multi-agent environment. *Annals of Mathematics and Artificial Intelligence*, 37(4):381–407, 2003.
10. A. Gabaldon. Programming hierarchical task networks in the situation calculus. In Proc. of *AIPS'02 Workshop on On-line Planning and Scheduling*, Toulouse, France, April 2002.
11. M. P. Georgeff and F. F. Ingrand. Decision making in an embedded reasoning system. In *Proc. of IJCAI'89*, pages 972–978, Detroit, USA, 1989.
12. K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. Meyer. A formal semantics for an abstract agent programming language. In *Proceedings of the International Workshop on Agent Theories, Architectures, and Languages (ATAL)*, pages 215–229, 1998.
13. K. V. Hindriks, Y. Lespérance, and H. J. Levesque. An embedding of ConGolog in 3APL. Technical Report UU-CS-2000-13, Dept. of Computer Science, University Utrecht, 2000.
14. M. J. Huber. JAM: A BDI-theoretic mobile agent architecture. In *Proceedings of the Annual Conference on Autonomous Agents (AGENTS)*, pages 236–243, New York, USA, 1999.
15. H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.
16. M. Ljungberg and A. Lucas. The OASIS air-traffic management system. In *Proceedings of the Pacific Rim International Conference on Artificial Intelligence (PRICAI)*, 1992.
17. D. Morley and K. L. Myers. The SPARK agent framework. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 712–719, New York, USA, 2004.
18. A. Pokahr, L. Braubach, and W. Lamersdorf. JADEX: Implementing a BDI-infrastructure for JADE agents. *EXP-In search of innovation (Special Issue on JADE)*, 3(3):76–85, 9 2003.
19. M. E. Pollack. The uses of plans. *Artificial Intelligence Journal*, 57(1):43–68, 1992.
20. A. S. Rao. Agentspeak(L): BDI agents speak out in a logical computable language. In LNCS, vol. 1038, p. 42–55. Springer, 1996.
21. A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In *Proc. of KR*, pages 473–484, 1991.
22. A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In *Proc. of KR'92*, pages 438–449, 1992.
23. R. Reiter. *Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, 2001.
24. S. Sardina, G. De Giacomo, Y. Lespérance, and H. J. Levesque. On the semantics of deliberation in IndiGolog – From theory to implementation. *Annals of Mathematics and Artificial Intelligence*, 41(2–4):259–299, Aug. 2004.
25. S. Sardina and L. Padgham. Goals in the context of BDI plan failure and planning. In *Proc. of AAMAS*, pages 16–23, Hawaii, USA, May 2007. ACM Press.
26. M. Thielscher. FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming*, 5(4–5):533–565, 2005.
27. M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative & procedural goals in intelligent agent systems. In *Pro. of KR*, pages 470–481, Toulouse, France, April 2002.

Agent-Oriented Control in Real-Time Computer Games

Tristan M. Behrens

Department of Informatics, Clausthal University of Technology
Julius-Albert-Straße 4, 38678 Clausthal, Germany
behrens@in.tu-clausthal.de

Abstract. In this paper, we introduce a method to apply agent-oriented programming (AOP) to computer games. We distinguish between two different views: (1) *single-agent* and (2) *multi-agent*. While (1) consists of path planning (based on A^*) and potential fields (for obstacle avoidance), (2) is based on ad-hoc networks. They facilitate the communication between agents that are close to each other. Finally, we show how AOP can be used to implement agents based on the two views.

1 Introduction

Multi-agent systems programming is a promising software engineering paradigm. It is especially suited for the development of systems that have to operate in dynamic environments[7]. Examples for AOP[2, 7] languages are the BDI-based Jason[3] and 2APL[6]. Jason is an implementation of an AgentSpeak(L) extension and 2APL realizes an effective implementation of both declarative and imperative programming.

This paper contains a method to apply AOP as a means to implement artificial intelligence for entities in real-time computer games. We have the strong feeling that AOP languages are suitable to function as the high-level control of entities in games. Especially in the case of games in which the number of entities is massive.

Why are we interested in computer games as platforms for the application of AOP? First of all computer games are a challenging compromise between toy examples and real world applications. Furthermore computer graphics cease to be the main drive of the industry and it is highly probable that the focus will shift to the AI. Additionally many computer games make the cooperation of entities (optionally with a human player) desirable.

One goal of AOP is to control robots. The fact that robots are embodied agents and thus are situated in the real world, implies at least two requirements: the agent needs (1) sensors for perceiving its environment and (2) actuators for acting in the environment. Of course, both requirements can be challenges for researchers. Computer games – especially those of the newest generation – have complex game worlds and sophisticated physics, that make them quite realistic. It is straightforward to create sensors and actuators for agents that

are situated in a game world. Sensors can be used to query the game state and actuators can be used to update it. But how to perceive? There seem to be two extremes: (1) agents have direct access to the game-state and (2) agents perceive exactly the same data as the human player. The first extreme leads to (undesired) omniscience of the agents – they know for example about the positions of all the entities in the game world, whereas a human player might only be aware of the visible entities. That is something that is not desired in computer games, because it can quickly give rise to frustration on the side of the human player. The second extreme would require the application of (visual) recognition systems. Both extremes make it desirable to create *virtual sensors*, that simulate visibility (and audibility et cetera) for the agents.

In our paper we propose

- a *single-agent view* based on using A* for high-level pathfinding and potential fields for low-level obstacle avoidance,
- a *multi-agent view* based on ad-hoc nets that facilitate the communication of agents respecting spatial information, and
- an implementation based on the AOP language 2APL.

In the second section we will motivate AI for computer games. The third section explains our approach. We then elaborate on our implementation. Finally we discuss similar work and conclude with future work.

2 Computer Games

In a typical computer game architecture, three integral components can be distinguished (see Fig. 1):

- the *game world simulator* that manages the *game-state*,
- the *visual/accoustic renderer* that renders the *current* game-state, and
- the *controllers* that allow to manipulate the *entities* in the game world.

The core of a computer game is usually the *game world simulator*. The game world simulator contains the *game state*, which consists of the game's entities (player, obstacles, items, opponents et cetera) and the environment in which the objects are situated, and the *game state transformer* which lets the game state evolve over time. A typical example for an integral component of a game state transformer is a *physics engine* which moves the entities in accordance to given laws of physics and deals with collision detection.

Another important component is the *visual/accoustic renderer*, which is responsible for the optical and aural representation of the game state for the human player.

Finally a set of *controllers* manipulates the entities in the game world. Controllers can be employed both by the human player and the computer player.

All three components constitute the axes of game-development. The bulk of game history was characterized by advances in computer graphics. The renderer has been the main focus of attention for decades. Now, at the dawn of realistic

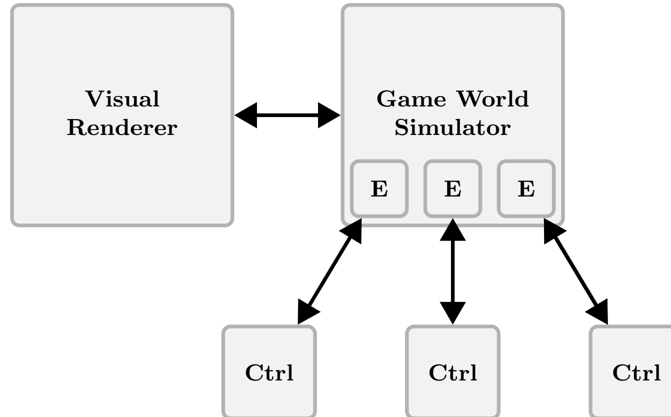


Fig. 1. The typical game architecture, with the tree components visual renderer (computer graphics), the game world simulator (physics) and the controlers (artificial intelligence).

and hyperrealistic graphics, it is probable that the focus will shift to artificial intelligence, which has been quite neglected in the past.

Also, it is an observable trend that game AI is more and more applied to creating teams of entities. Entities cooperate and coordinate their actions in order to beat the human player and entities form teams to *support* the human player as well. This is where the social component of agents come in handy. Here cooperation is necessary if the summation of the agents capabilities lead to the success of the whole team.

The state-of-the art of game AI is the combination of *finite state machines* and *pathfinding*[17]. A state machine represents the state of an entity and distinguishes between different behaviors.

3 Our Approach to Real-Time Strategy Games

Although our approach is aimed at a general class of computer games – in which the game world is populated by many entities that are controlled by some kind of AI – we will focus on *real-time strategy* (RTS) games in this paper. Real-time strategy games are not turn-based and usually incorporate some of the following concepts:

- **exploration:** the map is usually unknown territory at the beginning of each mission.
- **resource-gathering and -management:** resources are located in the environment and have to be gathered, usually in order to gain credits.
- **base building:** in order to establish and defend one's position in the territory and to produce armed units, a base has to be set up.

– **combat-oriented action:** usually the main goal is to overpower an opponent that is situated in the environment as well.
– **abstract unit control:** armed units are not controlled directly. Instead they are given orders that they act out.
– **technological development:** technological advances support performing the previous concepts.

The main focus of the RTS gameplay lies in base building and combat, usually in that sequence. Mission goals can be the following:

– complete destruction of the forces of the enemy,
– selective destruction of the opponent's structures,
– special operations, and
– object defense.

RTS games are usually quite complex. They feature a plethora of different armed units with different and task-dependent features, as well as a multitude of buildings with different functions. We omit, for our purposes, many of the features that increase the fun and excitement for the player but hinder the analysis from the AI perspective, and concentrate on the most interesting aspects instead. In this paper we will only focus on navigation and interaction.

We will use agents programmed in an AOP language to control entities in a game world. We will employ A^* for navigation, potential fields for collision avoidance and ad-hoc networks for communication Two views will be discussed:

– the *single-agent view* deals with the single agent and its interaction with the environment, and
– the *multi-agent view* deals with the coordination between several agents.

3.1 Single-Agent View

The core of this section is a navigational approach that steers entities through a game world. Agents distinguish two views of navigation: a *global view* and *local view*. The global view is the application of the *A^* algorithm* for finding shortest paths. The local view uses *potential fields* for collision avoidance. The global view dictates which location to go next, the local view takes into account all the visible obstacles in order to avoid bumping into them on the way to the goal.

A very good analogy to potential fields is the idea of an electron moving through an electromagnetic field of non-uniform structure. The moving electron would interact with the sources of the field by being diverted through forces of attraction and repellence. This idea can be applied to navigating entities in a game as well. Imagine an entity having a certain goal to reach. Above that the entity should not bump into obstacles. It is easy to model the goal as an attractive and all obstacles as a repelling field each. The combination of all the fields would then steer the entity to the goal on an obstacle avoiding path.

A potential field is a function $f \in \mathbb{R} \times \mathbb{R}^{\mathbb{R} \times \mathbb{R}}$ that maps a two-dimensional vector to another one. Usually the input-vector represents a position on the Euclidean plane and the output vector a force that is effective at that position.

Example 1 (potential fields). Any function

$$f_{gauss} : [x, y] \mapsto \frac{[x - x_0, y - y_0]}{|| [x - x_0, y - y_0] ||} \cdot a \cdot \exp\left(-\frac{|| [x - x_0, y - y_0] ||^2}{2s^2}\right)$$

is called a *Gaussian repeller* potential field. The constant vector $[x_0, y_0]$ represents the *center*, and the constants a and s represent the *amplitude* and the *spread* of the field respectively.

The repelling force is strongest at the center and steeply falls off, converging to 0. An entity approaching a Gaussian repeller will be affected once it gets close to that force. The amplitude a determines the maximum strength of the force. The spread s determines the width of the Gaussian bell and thereby the range of influence of the field.

Another potential field is the sink attractor:

$$f_{sink} : [x, y] \mapsto \frac{[x - x_0, y - y_0]}{|| [x - x_0, y - y_0] ||} \cdot g_{sink}(x, y)$$

with

$$g_{sink} : [x, y] \mapsto a \cdot \exp\left(-\frac{|| [x - x_0, y - y_0] ||^2}{2s^2}\right) - g \cdot || [x - x_0, y - y_0] || - a$$

The constant vector $[x_0, y_0]$ represents the *center*. The constants a and s represent the *amplitude* and the *spread* respectively. The constant g represents the grade. In the sink attractor the attractive force is stronger the farther away the target is. It is the combination of a conical potential field and a Gaussian one. Fig. 2 shows the two potential fields as vector fields in the Euclidian plane.

So, how do we combine multiple potential fields in order to come up with an overall potential field for each agent? The overall potential field of an agent ag is a set $F_{ag} := \{f_{ag,1}, \ldots, f_{ag,n}\}$ of potential fields. The overall force at a given position is $F_{ag}(x, y) := \sum_{f_{ag,i} \in F_{ag}} f_{ag,i}(x, y)$, which is just the sum of all forces that have an effect at that position. An agent would firstly come up with the set of all fields that affect him and then follow the calculated force vector.

How can we use the two exemplary potential fields? If the agent has the goal of being at a given position, obstacles (static ones and enemy units) could be associated with a repeller each, and the goal position could be represented by an attractor. The sum of all potential fields that affect the agent is the overall force field. The agent affected by that force-field would move away from obstacles towards the goal position. If on the other hand the agent should show the behavior of engaging a target unit, the target unit would be associated with an attractor as well.

We apply the A* algorithm on top of the potential fields. The main problem of the potential fields method is that it easily gets stuck in local optima, never reaching the goal. The A* algorithm is an informed search algorithm for finding shortest paths in graphs[16]. The algorithm it is quite common in computer games development. How do A* and potential fields combine? First of all the

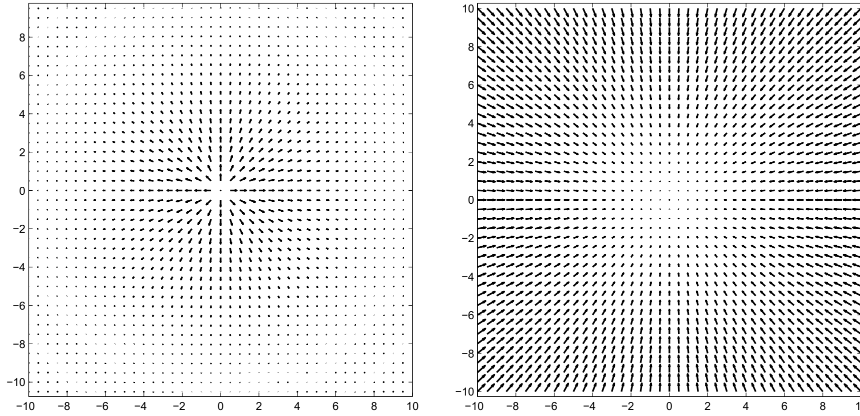


Fig. 2. Two potential fields shown as vector fields. The left one shows the Gaussian repeller. The upper right one shows the sink attractor.

agent computes a shortest path using the A* algorithm. Then it associates an attractive field with the first node in the path. The agent then follows its overall potential field. Once the node is reached it is removed from the path and the agent continues with the next one until the goal is reached.

The following algorithm implements the proposed navigational method:

Algorithm 1 Simple Navigation Algorithm

$P := calculatePath()$; //invoke A*
while $P \neq \emptyset$ **do**
  $p := removeFirst(P)$; //get and remove the first element
  $f := toField(p)$; //convert to field
  $F := F \cup \{f\}$; //add to overall field
  $moveTo(p, F)$ //move to p following F
  $F := F/\{p\}$; //remove from overall field
end while

3.2 Multi-Agent View

In this section we will focus on the multi-agent view of our approach. It is in its essence the communication between agents, based on *ad-hoc nets*. Ad-hoc nets are communication networks, whose topology is highly dynamic. The network structure changes over time, the nodes do not rely on a given (hierarchical) topology. Usually the topology of an ad-hoc net is determined by the nodes' position in space and their communication ranges. Nodes only communicate

with their nearest neighbors. If data has to be transmitted farther, it is forwarded through the net. Each node itself functions as a router that forwards unrelated messages.

A common type of ad-hoc nets are *MANets*, short for *Mobile Ad-hoc Networks*. They are self-configuing networks, consisting of mobile routers connected by wireless links. The topology is arbitrary and the nodes are allowed to move and arrange themselves arbitrarily. A specialization are *VANets*, short for *Vehicular Ad-hoc Networks*. Here nodes are either vehicles or nearby traffic equipment. The goal of VANets is to provide safety and comfort for passengers by providing communication between vehicles and equipment. This can be used for collision warnings and traffic monitoring. In VANets vehicles usually move in an organized fashion. *Mobile Sensor Networks* are wireless networks that are composed of autonomous sensors. They were originally motivated by military applications and are now used in civilian areas of application.

So, why would we use ad-hoc networks? Reasons are to cut down communication load and exploiting spatial information. Agents that are close to each other would be better suited to share information than agents that are far away. Using ad-hoc nets for communication nevertheless does not stop an agent from communicating with an other one that is far away. The first agent would just send a message using a flooding algorithm. We belief that this is a good compromise for our application to computer games: cooperation of agents is facilitated by spatial proximity.

We have decided to use triangulations for the network-structures. A triangulation is a graph that is planar and has the maximum number of edges. Triangulations cut down the complexity of the structure immensely.

In our approach each single agent is able to generate its own *local* ad-hoc net, taking into account all the other agents in the communication range. These agents are associated with nodes in the network. A graph is generated employing a simple triangulation algorithm that firstly generates a complete graph and then removes all long edges that intersect with others, which will finally result in a triangulation. Fig. 3 shows a possible triangulation of a set of agents that have arbitrary positions.

4 Implementation

Our implementation is based on the 2APL platform[6], that has kindly been made available to us by the developers in Utrecht. 2APL agents are cognitive agents and consist of *beliefs*, *goals*, *actions*, *plans* and *rules*. Beliefs model the information the agent has about its world, goals denote the state the agent wants to achieve, actions are the agent's means to manipulate the world, and rules – if applied – instantiate plans based on the agent's current goals and beliefs. We have extended the 2APL platform a bit to serve our needs. One thing that we did was the introduction of two special actions that allow the addition and the removal of agents during runtime. In its current implementation the 2APL platform only allows for multi-agent systems that are static in that respect. This

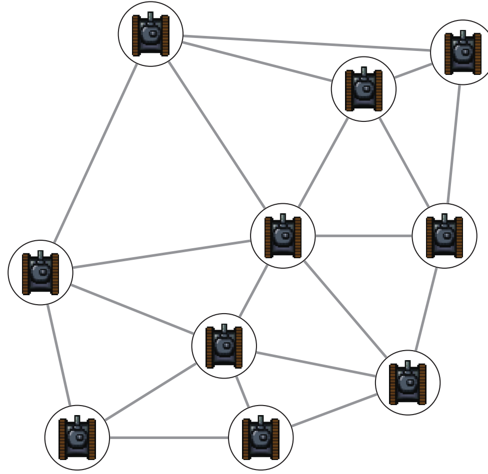


Fig. 3. A triangulation between of set of agents. Agents only communicate directly with direct neighbors. Other agents can be reached by flooding the network.

extension deemed to be necessary after the analysis that many computer games are highly dynamic in respect to the number of entities. An agent is now able to create one or several agents by loading the specification from a file. Also each agent is able to terminate itself.

Our scenario's game world (the physical model) is an extension of the standard 2APL environment. Agents are capable of interacting with the environment via two ways: *external actions* and *external events*. External actions allow the agents to contribute to the state-change of the environment and events are issued by the environment to inform the agent.

The 2APL platform is extremely flexible when it comes to the *execution model* of the agents. In its standard implementation the multi-agent system is executed in a multi-threaded way, each agent is executed in its own thread. In computer games development the tradition has been established to implement the whole project in a single thread[17]. Using a single thread makes it easy to keep the frame-rate (number of scenes rendered per second) on an adequate level that does not ruin the player's user-experience. We have refrained from following that approach. Instead we have two threads: the game world thread and the artificial intelligence thread. The first thread implements the evolution of the game world, it moves objects in real time, manages collisions, implements the game's logics and renders the scene. The second thread evolves the agents, each agent is allow to deliberate by one step in each cycle of the thread.

4.1 Scripting Language

In our approach 2APL can be considered a kind of "scripting language" for agents on top of a custom API. Fig. 4 shows the architecture. The agent layer represents top-level beliefs, goals and plans. The data layer contains the potential field and the ad-hoc net. The physical layer represents the agent's embodiment in the environment, including sensors and actuators.

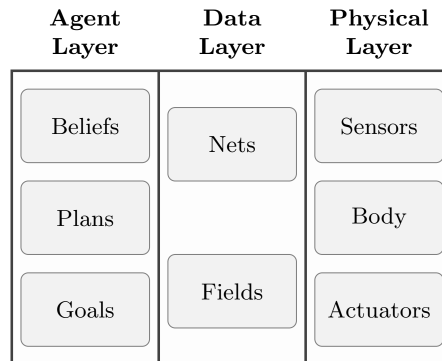


Fig. 4. A closer look at the relationships between agents, entities and the game world. An entitiy is a kind of vessel for sensors/actuators, potential fields and ad-hoc nets.

Table 1 shows some exemplary external actions, that can be used to act and perceive in the game world and to manipulate the potential field and the ad-hoc net. Based on that API, different behaviors can be implemented. We will now have a look at an example for navigating through the environment.

Example 2 (path-finding and -following). Consider this code fragment:

```
PG-rules:
  beAt(X,Y) <- true | {
    @env( getPath(X,Y), P); // invoke A*
    followPath(P)  // follow the path
  }

PC-rules:
  followPath(Path) <- true | {
    B(Path = [pos(GX,GY)|R]); // get the next goal position
    @env( addAttractor(GX,GY), _ ); // add attractor
    @env( followPotentialField(),); // be affected by the field
    B( isAt(GX,GY) ); // wait for goal to be reached
    @env( removeAttractor(GX,GY), _ ); // remove the attractor
    if B( not(R = [])) then // continue with next goal position
    {
      followPath(R)
    }
  }
```

External Action	Description
getPath(X,Y)	invokes A* and returns a shortest path
senseMovingObstacles()	returns a list of moving obstacles that are visible
senseStaticObstacles()	return a list of static obstacles that are visible
addAttractor(X,Y,R)	adds an attractor to the potential field
addRepeller(X,Y,R)	adds a repeller to the potential field
removeAttractor(X,Y)	removes an attractor from the potential field
removeRepeller(X,Y)	remove a repeller from the potential field
getField()	returns the overall potential field
getAttractors()	returns all attractors in the potential field
getRepellers()	returns all repellers in the potential field
followPotentialField()	moves the entity along the potential field
getNet()	calculates and returns an ad-hoc net
getNearestNeighbors()	returns the nearest neighbors

Table 1. Some external actions.

Rules in 2APL are used to instantiate plans in respect to beliefs, goals and events. Here we have two rules. The first rule is a *PG-rule*. It instantiates the plan if the agent has the goal to be at a certain position (`beAt(X,Y)`). The plan consists of two statements. At first a path to the goal is calculated by invoking the external action `getPath`. Then the path is followed by raising the procedural event `followPath`. The second rule is a *PC-rule*. The plan is instantiated once an event `followPath(P)` has been raised. The first statement of the plan separates the path into its first element `pos(GX,GY)` and the rest `R`. The coordinates of that first element are then used to add an attractor to the potential field by invoking the external action `addAttractor`. The agent then follows the potential field executing `followPotentialField`. Once the first element of the current path is reached (once the agent believes being at the desired position) the attractor is removed via `removeAttractor` and the rest of the list is processed recursively.

This example shows a very simple navigational routine. A path is calculated and then followed by adding and removing attractors to and from the potential field. An agent executing that behavior does not take obstacles into account and thus would bump into them. Such a stubborn behavior might be desired in a computer game for the user experience. Nevertheless we improve the example a bit.

Example 3 (obstacle avoidance). Consider the following code:

```
PG-rules:
  avoid(obstacles) <- true | {
    @env( senseMovingObstacles(), M); // sense obstacles
    addRepellentFields(M) // add fields recursively
  }
```

```
PC-rules:
  addRepellentFields(Units) <- true | {
    B( Units = [obstacle(X,Y)|R] ); // get the first obstacle
    @env( addRepeller(X,Y), _ ); // add a repellent force
    if B( not(R = [])) then // add the rest if rest not empty
    {
      addRepellentFields(R) // proceed recursively
    }
  }
```

The first rule implements the following: if the agent has the goal of avoiding the visible obstacles the respective plan is instantiated. Firstly the moving obstacles are sensed and secondly they are added as repellers to the potential field. The second rule implements the addition of the repellers in a recursive fashion. The first element is determined and added as a a repeller. After that the rest of the list is processed until the list is empty.

What is the benefit from our approach of layering AOP in top of entities? We preserve the agents' autonomy. This way agents can reason about the potential fields and ad-hoc nets and decide their behavior according to the results. Agents could for example decide to omit certain repellers or add attractors. They could for example decide to only take friendly entities into account as obstacles. This would lead to a behavior of avoiding friendly entities and bumping into enemies. Furthermore agents can decide when to update the data-structures. An agent might chose to not update its potential field if it is busy doing something more important. An agent could also manipulate the representation of the potential field. For example if it comes to the conclusion that an obstacle is extremely threatening, its repelling force could be increased. Our approach allows to easily implement and combine different behaviors to enhance the user experience.

To conclude this section we will elaborate on an other issue that has been left undiscussed until now: human-agent interaction. How does the human player interact with the entities under his command? The environment allows the selection of one or several entities. The commands traditional to RTS games ("go there", "attack this", "guard that") are interpreted and sent to the agents as messages. To that end we use to the message-passing facilities underlying 2APL.

You can see a screenshot of our prototype in Fig. 5.

4.2 Entity Association Extension

We have shown how a 2APL agent can be implemented to control an entity in a game-world. We have associated one agent with one entity. Now we will extend that by associating *one agent* with *several entities*.

Why would it make sense to associate on agent with several entities? In our opinion it would be beneficial to do so when dealing with groups of entities. If a group should be used to fulfill a specific goal, a single agent could be employed to steer and coordinate the entities. For example if a group should engage a single target, the entities would have similar plans to reach that goal. It would make sense to have a single agent manage the execution of such a plan that is shared by the entities. This approach would be the basis for an AOP-control of

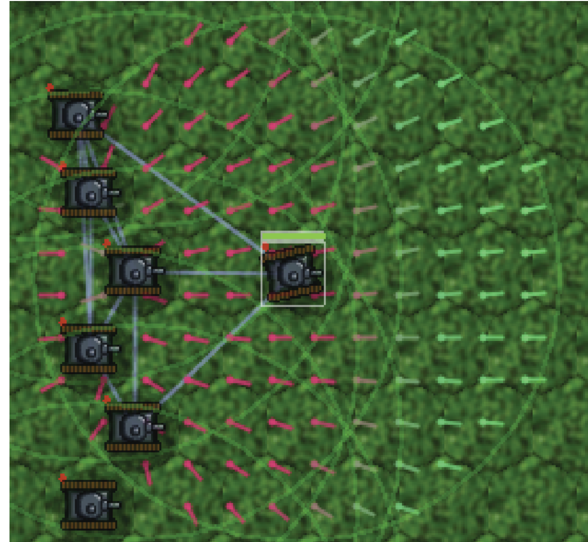


Fig. 5. A screenshot of our prototype. The arrows indicate the potential field in sensor range. The blue lines denote the triangulation.

a massive number of entities, in the sense of having few agents control many entities.

How do we realize that idea? To associate one agent with several entities we have to do two things: 1) extending the physical layer of our architecture (see Fig. 4) to respect several bodies/sensors/actuators, and 2) extending the API with new actions that the agent can execute to control the associated entities. An agent would still have a potential field and an ad-hoc net, but now several entities. Table 2 shows a second set of exemplary actions, most of them are related to the actions in Table 1. To make the agent aware of its associated entities the action `getEntities` can be used. The actions `senseMovingObstacles` and `senseStaticObstacles` return the obstacles that are visible to the given set `E` of entities. The action `followField(E)` on the other hand lets the specified entities `E` follow the potential field of the agent. It is important to have most of the function parametrized in respect of the entities in order to allow the agent to decide which of the associated entities to take into account.

Finally, we elaborate how associations are created and modified. We define the initial association to consist of one agent, the *default agent*, and all entities. This allows us to define standard behaviors for (idle) entities that are implemented by the default agent. Once a group of agents is created the association between the entities and the default agent is transferred to an other agent that is supposed to control the group. Therefore the action `transferEntities(E,A)` is used.

External Action	Description
`getEntities()`	returns a list of associated entities
`senseMovingObstacles(E)`	returns a list of moving obstacles that are visible to the entities `E`
`senseStaticObstacles(E)`	return a list of static obstacles that are visible to the entities `E`
`followField(E)`	lets the entities `E` follow the potential field
`transferEntities(E,A)`	transfers the entities `E` to the agent `A`

Table 2. Some external actions for case that one agent is associated with several entities.

5 Related Work

The potential fields methology was depeloped by Krogh[12] and Khatib[11] for obstacle avoidance control. Krogh borrowed the terminology from analytical mechanics. He determined a collision free path to transfer a system to a work-space. Khatib concentrated on a real-time obstacle avoidance approach for manipulators and mobile robots. Arkin in his book[1] describes the potential fields methodology as a functional mapping from stimuli to motor-responses, in order to encode a continuous navigational space through the sensed world. Massari et al. in their paper[14] use potential fields to steer planetary rovers. All of the mentioned authors intended their research to be applied to real-world applications. Furthermore all of them did not take the multi-agent perspective into account.

Hagelbäck and Johansson in their very good papers[10, 9] illustrate an application of potential fields on the research platform ORTS[4]. The main difference between their approach and ours is that they rely on complete information, whereas we work with incomplete information. They have full access to the state of the world and we rely on the (limited) local view of the agents. The difference is the reason why they can afford to calculate a discretized potential field for the complete map and we are forced to use local ones. Our approach is suited to deal with group of agents with different goals. The main aim of Hagelbäck and Johansson seems to be AI that wins, we are more interested in the user experience. They have no high-level path planning, we employ A^*. Also, we use ad-hoc nets as the basis for coordinating agents. Finally we resort to AOP for the high-level control of the entities.

Conceicao et al. introduce in their paper[5] a realistic mobile connectivity model for vehicular sensor networks in urban environments. They focus on the evolution of the average node degree in the graph and provide a characterization of the connectivity of a vehicular sensor network operating in an urban environment. Muhammad[15] presented a fully distributed algorithm to compute a planar graph and a geometric routing algorithm. His results are based on idealized unit disk graph model, that assumes that the communication range is the same for all the nodes. We do not share that assumption. The algorithm that we

use to generate planar graphs is a simple one. Muhammad, on the other hand, generates Delaunay triangulations.

Davies and Mehdi present a prototype application[8] that implements a BDI agent system (Jadex) within the first person shooter Unreal Tournament, based on the GameBots and JavaBots technology. Their agents are capable of creating an internal representation of the three-dimensional world, navigate in it and show some basic behaviors. Our approach differs in two ways: we concentrate on a completely different world model and we are interested in a massive number of agents/entities.

In his book[17] Schwab summarizes the state of the art of computer games AI that he rightly distinguishes from academic AI that has been applied in the industry.

6 Conclusion and Future Work

In this paper we have explained an approach to AI in computer games that feature a lot of entities. To that end we have used the RTS scenario. RTS games usually contain a lot of entities that are controlled by issuing orders. AOP serves as a high-level control of the entities. The entities navigate using the A^* algorithm for path planning and potential fields for obstacle avoidance. Furthermore we use ad-hoc networks as a means for communication and cooperation.

We are interested mostly in the two extremes of using agents to control entities in computer games. One extreme is our approach. We associate one agent with exactly one entity in the game world. The other extreme is to have one agent that steers all entities. We would like to investigate the "in between" and find out, when does it make sense to associate one agent with several entities. In our approach this would make sense especially if the agents have a common (pathfinding) goal. We plan to associate one agent with one potential field, one ad-hoc net and several entities and take a look at how many entities should be associated.

Furthermore we would like to examine deeper how ad-hocs net can be used. How can they be used to coordinate strategies and tactics? Would it make sense to use them to negotiate collision avoidance maneuvers? How can they help to facilitate team-work in the planning sense?

Finally we think it would be fruitful to apply our approach to other games that feature a massive number of entities. God games, in which entities are steered indirectly by influencing the environment, come to mind immediately.

References

1. Ronald C. Arkin. *Behavior-Based Robotics (Intelligent Robotics and Autonomous Agents).* The MIT Press, May 1998.
2. R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Programming Multi Agent Systems: Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies and Simulated Organizations.* Springer, Berlin, 2005.

3. Rafael H. Bordini, Michael Wooldridge, and Jomi Fred Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.
4. Michael Buro. ORTS: A hack-free RTS game environment. In *Computers and Games*, pages 280–291, 2002.
5. Hugo Conceiç ao, Michel Ferreira, and Jo ao Barros. On the urban connectivity of vehicular sensor networks. In *DCOSS '08: Proceedings of the 4th IEEE international conference on Distributed Computing in Sensor Systems*, pages 112–125, Berlin, Heidelberg, 2008. Springer-Verlag.
6. Mehdi Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.
7. Mehdi Dastani, Amal El Fallah-Seghrouchni, Alessandro Ricci, and Michael Winikoff, editors. *Programming Multi-Agent Systems, 5th International Workshop, ProMAS 2007, Honolulu, HI, USA, May 15, 2007, Revised and Invited Papers*, volume 4908 of *Lecture Notes in Computer Science*. Springer, 2008.
8. N.P. Davies and Quasim Mehdi. BDI for intelligent agents in computer games. In *Proceedings of CGAMES'2006*, 2006.
9. Johan Hagelbäck and Stefan J. Johansson. The rise of potential fields in real time strategy bots. In *AIIDE*, 2008.
10. Johan Hagelbäck and Stefan J. Johansson. Using multi-agent potential fields in real-time strategy games. In *AAMAS (2)*, pages 631–638, 2008.
11. O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. volume 2, pages 500–505, 1985.
12. B. Krogh. A generalized potential field approach to obstacle avoidance control. 1984.
13. Michael Van Lent, John Laird, Josh Buckman, Joe Hartford, Steve Houchard, Kurt Steinkraus, and Russ Tedrake. Intelligent agents in computer games. In *In Proceedings of The Sixteenth National Conference on Artificial Intelligence*, pages 929–930. AAAI Press, 1999.
14. M. Massari, G. Giardini, and F. Bernelli-Zazzera. Autonomous navigation system for planetary exploration rover based on artificial potential fields. In *Proceedings of Dynamics and Control of Systems and Structures in Space (DCSSS) 6th Conference*, 2005.
15. Rashid Bin Muhammad. A distributed graph algorithm for geometric routing in ad hoc wireless networks. *JNW*, 2(6):50–57, 2007.
16. S. J. Russell and Norvig. *Artificial Intelligence: A Modern Approach (Second Edition)*. Prentice Hall, 2003.
17. Brian Schwab. *AI Game Engine Programming (Game Development Series)*. Charles River Media, Inc., Rockland, MA, USA, 2004.

An Architecture for Multiagent Systems

An Approach Based on Commitments

Amit K. Chopra[1] and Munindar P. Singh[2]

[1] Università degli Studi di Trento akchopra.mail@gmail.com
[2] North Carolina State University singh@ncsu.edu

Abstract. Existing architectures for multiagent systems emphasize low-level messaging-related considerations. As a result, the programming abstractions they provide are also low level. In recent years, commitments have been applied to support flexible interactions among autonomous agents. We present a layered multiagent system architecture based on commitments. In this architecture, agents are the components, and the interconnections between the agents are specified in terms of commitments, thus abstracting away from low level details. A crucial layer in this architecture is a commitment-based middleware that plays a vital role in ensuring interoperation and provides commitment-related abstractions to the application programmer. Interoperation itself is defined in terms of commitment alignment. This paper details various aspects of this architecture, and shows how a programmer would write applications to such an architecture.

1 Introduction

An *architecture* is an abstract description of a system. The fundamental idea of an architecture is that it identifies *components* and their *interconnections*. An *open* architecture is one that emphasizes the interconnections, leaving the components unspecified except to the extent of their interconnections. In this manner, an open architecture yields systems whose components can be readily substituted by other components.

When we understand multiagent systems from the standpoint of architecture, it is clear that the components are *agents* (or, rather, abstractly *roles*). Traditionally, the interconnections have been modeled in operational terms derived from an understanding of distributed systems. Consequently, the multiagent systems that result are over-specified and behave in an inflexible manner. With such systems, it is difficult to accommodate a richer variety of situations.

For concreteness, we consider cross-organizational business processes as an application of multiagent systems that provide the happy mix of significance and complexity to demonstrate the payoff of using the proposed approach. The last few years have developed compelling accounts of the fundamental autonomy and heterogeneity of business partners, and the concomitant need to model these partners' interest. The related studies of interaction protocols hint at how we might engineer multiagent systems in such settings [1–3]. However, the relationship of protocols with architectures has not yet been adequately worked out.

We make a fresh start on multiagent systems via an architecture. Once we realize that we would only consider the components as agents understood flexibly, the associated interconnections must inevitably be the business relationships between the agents. One can imagine that some notional business value flows across such relationships, just as data flows over the traditional connectors of distributed computing. Thinking of the business relationships as interconnections yields an architecture for what we term *service engagements* [4].

The above architecture is conceptual in nature. Two natural questions arise: what programming abstractions does the architecture support, and how may we operationalize it over existing infrastructure, which is no different from that underlying traditional approaches. Answering the above questions is the main contribution of this paper.

1.1 Middleware: Programming Abstractions

From a top-down perspective, an important layer of any architecture is middleware. Middleware supports programming abstractions for the architecture in a way that ensures interoperability between components in the architecture. A relatively simple middleware is one that provides reliable message queuing services, freeing the programmer from the burden of, for example, implementing persistent storage and checking for acknowledgments. These days, reliable message queuing is just one of many abstractions supported in enterprise middleware. In cross-organizational business processes, the common middleware is centered on the abstractions of messaging. The resulting architectural style is termed the *Enterprise Service Bus (ESB)*. ESBs emphasize messaging abstractions and patterns—for example, Apache Camel supports the enterprise integration patterns in [5]. Further, ESBs support an event-driven architecture so as to promote loose coupling between business applications. ESBs provide various kinds of translation services, routing, and security, among other things, thus saving the application programmer a good deal of repetitive effort. Some ESB implementations, such as provided by Oracle, also support business protocols such as RosettaNet [6].

Ideally, middleware should offer abstractions that follow closely the vocabulary of the domain. ESBs purport to support business applications; however, they lack business-level abstractions. The abstractions they support, e.g., for RosettaNet, involve message occurrence and ordering but without regard to the meanings of the messages. Thus RosettaNet can be thought of as a protocol grammar. Other protocols, e.g., Global Data Synchronization Network (GDSN) [7], would correspond to alternative grammars. Each grammar is arbitrary and its correctness or otherwise is not up for consideration.

Figure 1 shows the conceptual arrangement of a service-oriented architecture based on such ESBs. Programmers design business *processes* (for example, in BPEL) based on a public interface specification (for example, in WS-CDL or based on a protocol such as RosettaNet). Messaging-based middleware, such as described above, hides the details of the infrastructure from process programmers.

1.2 Overview of Approach

We assume a conventional infrastructure based on messaging, such as is already made available by middleware such as the Java Messaging Service and specified in the emerg-

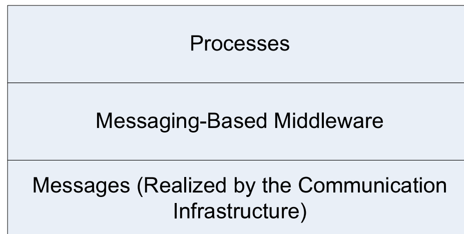


Fig. 1. Current enterprise middleware, conceptually

ing standard known as the Advanced Message Queuing Protocol (AMQP) [8]. This infrastructure supports point-to-point messaging over channels that preserve pairwise message order and guarantee eventual delivery. It is important to emphasize that such infrastructure is commonly available in existing implementations.

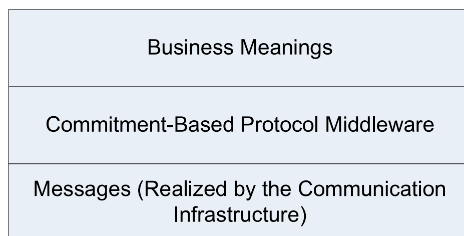


Fig. 2. Commitment middleware, conceptually

The essential idea underlying our approach is that we can thus view system architecture at two levels of abstraction: business and infrastructure. The business level deals with meaning whereas the infrastructure provides the operationalization. Accordingly, we view the function of middleware to bridge this conceptual gap. Figure 2 shows that our middleware lies in the middle between meaning and messaging. In our approach, business meaning is expressed in terms of commitments. Commitments arise in virtually all cross-organizational business applications. Thus, reasoning about commitments would be applicable to all of them. Commitments underlie two correctness criteria: *compliance* and *alignment*. Agents are compliant as long as they discharge their commitments; such a notion of compliance naturally takes into account agents' autonomy. Agents are aligned as long they agree on whatever commitments as may result from their communications. Alignment is, in fact, a key form of business interoperability [9, 10].

The proposed middleware provides commitment-based abstractions. The middleware supports not only the basic commitment operations [11], but also high-level patterns that build on the commitment operations. The middleware ensures that if applica-

tions are determined interoperable at the level of business meaning, then infrastructural-level concerns such as asynchrony do not break the interoperability.

The rest of this paper is organized as follows. Section 2 describes commitments formally, what alignment means, and why misalignments occur. Some misalignments can be detected at the level of business meanings by a static analysis of the interfaces, whereas others that occur due to the nature of distributed systems must be prevented by careful design of the middleware. Section 3 describes an architecture based on commitments. It describes the components and the interconnections and the layers in the architecture. Section 4 describes a sample set of useful patterns that the middleware supports. Section 5 discusses the relevant literature.

2 Commitment Alignment

Interoperability among participants means that each participant fulfills the expectations made by the others. To understand an architecture, it is important to understand what interoperability in the architecture means. In our approach, an agent represents each participant, and the expectations of an agent take the form of commitments. Existing work on service interoperability treats expectations solely at the level of messages [12–14].

Let us explain how commitments yield expectations. A commitment is of the form $\mathsf{C}(debtor, creditor, antecedent, consequent)$, where $debtor$ and $creditor$ are agents, and $antecedent$ and $consequent$ are propositions. This means that the debtor commits (to the creditor) to bringing about the consequent if the antecedent holds. For example, $\mathsf{C}(EBook, Alice, \$12, BNW)$ means that EBook commits to Alice that if she pays $12, then EBook will send her the book *Brave New World*. Agents interact by sending each other messages. The messages have meanings in terms of how they affect the agents' commitments toward each other. For example, an offer message from EBook to Alice may bring about the aforementioned commitment.

Now imagine that at some point in their interaction, Alice infers that EBook is committed to sending her the book she paid for, but EBook infers no such commitment. Their interaction would break down at the level of business meaning. In other words, Alice and EBook would not be interoperable. In general, a key requirement for interoperability is that the interacting agents remain aligned with respect to their commitments. Commitment alignment is a key form of business-level interoperability. Agents are aligned if whenever one agent (as creditor) infers a commitment from a second agent, the second agent (as debtor) also infers that commitment. If we can guarantee *a priori* that agents never—at no point during any possible interaction—get misaligned, only then the agents are interoperable.

In general, agents may get misaligned because of their *heterogeneity*, *autonomy*, and *distribution*.

Heterogeneity Agents may assign incompatible meanings to the messages they are exchanging. To be able to successfully interact, the agents must agree on what their communications count as. Heterogeneity is the cause of misalignment in Example 1.

Example 1. For Alice, an *Offer* message from EBook counts as a commitment from EBook to ship a book in return for payment. Whereas for EBook, *Offer* does not count as any such commitment; but an explicit *Accept* from Alice does. Thus, when EBook sends Alice an *Offer* message, Alice infers the commitment, but EBook does not—a misalignment.▌

Heterogeneity is addressed by statically analyzing if the interfaces of agents are compatible [9].

Autonomy Agent autonomy must be accommodated; however, accommodating autonomy is nontrivial. The reason is that autonomy operationally means that they are free to send messages. In turn, this means that communication between agents is asynchronous. Thus, in general, agents will observe messages in different orders. Since messages are understood in terms of their effects on commitments, the agents involved may become misaligned. This is the cause of misalignment in Example 2.

Example 2. EBook sends an *Offer* to Alice, where the offer means a commitment that if Alice pays, then EBook will send the book. Alice sends the payment (message) for the book. Concurrently, EBook cancels the offer by sending *CancelOffer*. Alice observes EBook's cancellation after sending the payment; so she regards it as spurious. EBook observes Alice's payment after sending its cancellation, so EBook considers the payment late. As a result, Alice infers that EBook is committed to sending her the book, but EBook does not infer that commitment. Thus, EBook and Alice are misaligned. ▌

An ideal approach to addressing the challenge of autonomy should work without curbing autonomy. In contrast, existing approaches to reasoning about commitments in distributed systems typically rely on some kind of synchronization protocol; synchronization, however, inhibits autonomy. Chopra and Singh [10] formalize the inferences made upon observing commitment-related messages in such a way that, in spite of autonomy, agents remain aligned.

Distribution In a distributed system, some agents may have more information about relevant events than others. This is the cause of misalignment in Example 3.

Example 3. Alice commits to Bob that if the sky is clear at 5PM, then she will meet him at the lake. At 5PM, Bob observes (a message from the environment) that the sky is clear, and therefore infers that Alice is unconditionally committed to meeting him at the lake. However, Alice does not know that the sky is clear, and therefore does not infer the unconditional commitment. Bob and Alice are thus misaligned. ▌

Chopra and Singh [10] state *integrity constraints*, which are constraints upon agent behavior necessary to handle distribution. Their constraints are of two kinds: (1) a debtor must inform the creditor about the discharge of a commitment, and (2) a creditor must inform the debtor about the detach of a commitment. One should not consider alignment until such information has been propagated.

2.1 Characterizing Alignment

A set of agents is aligned if in all executions, at *appropriate* points during their execution, if a creditor infers a commitment from its observations, the debtor also infers the commitment from its own observations [10]. An "appropriate" point in the execution of a multiagent system is given by consistent observations of the various agents where two additional properties hold. One, alignment may only be considered at those points where no message is in transit. Such points are termed *quiescent*. Two, alignment may only be considered at those points that are *integral* with respect to the stated information propagation constraints. The motivation behind the above properties is simply that it would surprise no one if two agents failed to infer matching commitments when they had made differing observations: either because some message was in transit that only its sender knew about or because some message was not sent, and some agent had withheld material facts from another.

2.2 Background on Commitments

A commitment is of the form $\mathsf{C}(x, y, r, u)$ where x and y are agents, and r and u are propositions. If r holds, then $\mathsf{C}(x, y, r, u)$ is *detached*, and the commitment $\mathsf{C}(x, y, \top, u)$ holds. If u holds, then the commitment is *discharged* and doesn't hold any longer. All commitments are *conditional*; an unconditional commitment is merely a special case where the antecedent equals $\top$. Singh [15] presents key reasoning postulates for commitments.

The commitment operations are reproduced below (from [11]). CREATE, CANCEL, and RELEASE are two-party operations, whereas DELEGATE and ASSIGN are three-party operations.

- CREATE(x, y, r, u) is performed by x, and it causes $\mathsf{C}(x, y, r, u)$ to hold.
- CANCEL(x, y, r, u) is performed by x, and it causes $\mathsf{C}(x, y, r, u)$ to not hold.
- RELEASE(x, y, r, u) is performed by y, and it causes $\mathsf{C}(x, y, r, u)$ to not hold.
- DELEGATE(x, y, z, r, u) is performed by x, and it causes $\mathsf{C}(z, y, r, u)$ to hold.
- ASSIGN(x, y, z, r, u) is performed by y, and it causes $\mathsf{C}(x, z, r, u)$ to hold.

Let us define the set of messages that correspond to the basic commitment operations. Let Φ be a set of atomic propositions. In the commitment operations, r is a DNF formula over Φ (for example, $(\phi_0 \wedge \phi_1) \vee (\phi_3 \wedge \phi_4)$), and u is a CNF formula over Φ (for example, $(\phi_0 \vee \phi_1) \wedge (\phi_3 \vee \phi_4)$) . $\mathit{Create}(x, y, r, u)$ and $\mathit{Cancel}(x, y, r, u)$ are messages from x to y; $\mathit{Release}(x, y, r, u)$ from y to x; $\mathit{Delegate}(x, y, z, r, u)$ from x to z; and $\mathit{Assign}(x, y, z, r, u)$ from y to x. Suppose $c = \mathsf{C}(x, y, r, u)$. Then $\mathit{Create}(c)$ stands for $\mathit{Create}(x, y, r, u)$. We similarly define $\mathit{Delegate}(c, z)$, $\mathit{Assign}(c, z)$, $\mathit{Release}(c, y)$, and $\mathit{Cancel}(c, x)$. $\mathit{Inform}(x, y, p)$ is a message from x to y, where p is conjunction over Φ. Observing an $\mathit{Inform}(p)$ causes p to hold, which may lead to the discharge or detach of a commitment.

Below, let $c_B = \mathsf{C}(\mathit{EBook}, \mathit{Alice}, \$12, \mathit{BNW})$; $c_G = \mathsf{C}(\mathit{EBook}, \mathit{Alice}, \$12, \mathit{GoW})$; $c_0 = \mathsf{C}(\mathit{EBook}, \mathit{Alice}, \$12, \mathit{BNW} \wedge \mathit{GoW})$. ($\mathit{BNW}$ stands for the book *Brave New World*; GoW stands for the book *Grapes of Wrath*).

3 Multiagent System Architecture

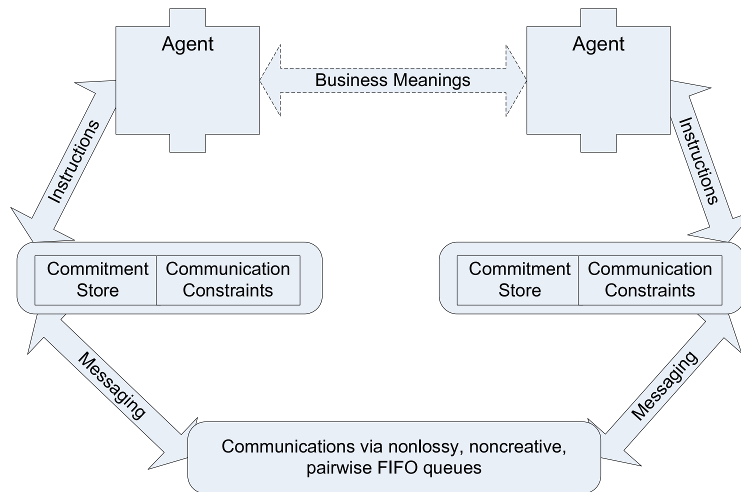


Fig. 3. Understanding Commitment-Based Architecture

Figure 3 shows our proposal for a multiagent system architecture. At the level of business meaning, the components are the agents in the system representing the interacting business partners. As pertains to programming using our architecture, at the top, we have agents and at the bottom the communication layer; the middleware sits in between. This layered architecture is characterized by three kinds of interfaces.

- At the business level, the interface is between agents and is expressed via meanings. The business analyst and the software developer who programs using commitments would think at this level (of business relationships), and would be unaware of any lower layer.
- At the implementation level, the interface is between our middleware and the communication infrastructure and is based on traditional messaging services. In a traditional distributed system, a software developer would need to think at this level. In our approach, only the implementor of our middleware thinks at this level.
- Between the agent and the middleware, the interface is largely in terms of instructions from the agent to the middleware: when an agent tells the middleware to apply a commitment operation or one of the additional patterns based on commitments such as *Escalate* (patterns described later).

Two nice features of our approach are that (1) the instructions use the same vocabulary as the business meanings and (2) we specify middleware that guarantees alignment as long as the instructions are limited to the commitment operations or patterns. Below, we describe each components (agents), interconnections (interfaces), and layers in detail.

3.1 Agents

Agents represent business partners. They provide and consume real-world services by participating in service engagements. The principal elements of interest in an agent are its interface and its reasoning engine.

Interface An agent's interface describes the messages it expects to exchange with other agents, along with the business meanings of such messages. Table 1 show a sample interface for Alice. The axioms in the interface are of the form Message means Meaning, where meaning is expressed as one of a small vocabulary of messages involving unit and composite operations on commitments. This vocabulary has a precise meaning in terms of their effects on the participants' commitments. For example, observing an offer from a merchant means observing a create message for the commitment corresponding to the offer.

Table 1. An example interface for Alice

$Offer(EBook, Alice, \$12, BNW)$ means $Create(EBook, Alice, \$12, BNW)$
$Accept(Alice, EBook, \$12, BNW)$ means $Create(Alice, EBook, BNW, \$12)$
$Reject(Alice, EBook, \$12, BNW)$ means $Release(EBook, Alice, \$12, BNW)$
$Deliver(EBook, Alice, BNW)$ means $Inform(EBook, Alice, BNW)$
$Pay(Alice, EBook, \$12)$ means $Inform(Alice, EBook, \$12)$

Notice that the interface does not contain some of the procedural constructs commonly found in interface description languages or protocols, such as sequence, choice, and so on. For example, it does not say, that upon observing an offer Alice has a *choice* between accepting or rejecting the offer—there is simply no need to say so. A rejection sent after Alice accepts the offer and EBook sends the book should have no effect—Alice should remain committed to pay. The formalization of commitments in [10] captures such intuitions, and makes the statement of procedural constructs in interfaces largely unnecessary. A second reason such constructs are introduced is to simply make the interaction synchronous. However, such constructs are rendered superfluous by the approach for reasoning about commitments in asynchronous settings [10]. Finally, threading constructions such as fork and join are clearly implementation details, and have no place in an interface. Of course, if an application demands a procedural construct, it could be introduced. For example, Alice may not trust booksellers and her interface might constrain delivery of books before payment. Alice will then be noninteroperable at the messaging-level with booksellers who require payment first; however, it would not affect alignment, that is, commitment-level interoperability.

As described earlier, misalignments arise when agents ascribe incompatible meanings to messages. An application programmer would specify an interface and publish it. Before interacting with other agents, the agent would presumably check for compatibility with the other agents.

Engine The engine drives the agent. It represents the private policies of the agent; these govern when an agent should pass an instruction to the middleware, how instruction parameters should be bound, how an agent should handle returned callbacks (described below) and so on. In fact, the engine is the place for all the procedural details. For example, Alice's policy may enforce a choice between accept and reject upon receiving an offer, or dictate that payment be sent only after receiving books.

Writing the engine is where the principal efforts of a programmer are spent. The implementation of the engine could take many forms. It could be a BPEL, Jess, JADE, or a BDI implementation such as Jason, for example. The details are irrelevant as long as it is consistent with reasoning about commitments.

From the programming perspective, the engine is coded in terms of the *meaning* of a message, not the message itself. In other words, the API that the programmer uses to interface with the middleware is in terms of commitment operations and other patterns built on top of the commitment operations. When the meaning concerns the sending of the message, the meaning may be thought of as an instruction (API) from the agent's engine to the middleware. The middleware, which is configured with the agent's interface, then sends the appropriate messages. Analogously, for an incoming message, the engine registers a callback with the middleware that returns when the commitment operation corresponding to the message has been executed. Thus, the programmer's API is a business-level one, one of the goals we set out to achieve.

3.2 Middleware

To relate meanings to messages, the middleware takes on the responsibility for representing and reasoning about commitments. The middleware consists of a commitment reasoner, maintains a commitment store, and is configured with communication constraints needed for the commitment operations and the further patterns. The middleware ensures that no misalignments arise because of autonomy and distribution.

Each agent's copy of the middleware is configured with the agent's interface (relating the agent's incoming and outgoing communications and their meanings, as in Table 1) so that the middleware may appropriately process messages and compute commitments. As described above, the middleware's interface with the agent is instruction and callback-based.

The commitment reasoner presents a query interface to the agent (specifically the agent's engine), which can be used to inquire about commitments in the store. The engine can use such a information to decide on a course of action. For example, Alice's policy might be such that she sends $Pay(Alice, EBook, \$12)$ only if $\mathsf{C}(EBook, Alice, \$12, BNW)$ holds.

The middleware maintains a *serial*, point-to-point communication interface with each other agent in the system through the communication layer. This means that an agent's middleware processes messages involving another particular agent—sent or received—one at a time. This is necessary to ensure consistency of the commitment store.

3.3 Communication Layer

The role of the communication layer is to provide reliable, ordered, and noncreative delivery of messages. Reliability implies that each sent message is eventually delivered; ordered implies that any two messages sent by an agent to another will arrive in the order in which they were sent, and noncreative means messages are not created by the infrastructure. Such a communication layer can be readily implemented by available reliable message queuing solutions.

3.4 Example Scenario

Going back to our purchase example, when Alice and EBook decide to participate in a purchase transaction, they would configure their respective sections of the middleware with the message meanings that they would entertain sending or receiving. Suppose EBook wishes to sell BNW to Alice. EBook computes this on internal grounds, such as excess inventory or the goal of making a profit. At the level of business meaning, EBook sends an offer to Alice. At the computational level, this is effected by EBook instructing its middleware to create the corresponding commitment. EBook's middleware then sends Alice the offer message. At the computational level, Alice's middleware receives the message from the communication layer, computes the corresponding commitment, and triggers Alice's callback on the creation of that commitment to return, in effect returning to the business level. Alice may reason on the commitment and may decide to accept the offer, based on her private considerations such as goals. Alice responds by accepting—by instructing her middleware to do so; and so on. In other examples, the more complex communication constraints would also apply.

4 Abstractions Supported by the Middleware

As mentioned before, the middleware supports sending notifications to debtors and creditors about detaches and discharges, respectively. The middleware also supports other integrity constraints critical to alignment. The middleware supports all commitment operations, including delegation and assignment, which are three-party operations, and guarantees that even in asynchronous settings, the operations occur without giving causing misalignments. Details are in [10]. Here, we discuss even high-level abstractions in the form of commitment patterns and additional forms of alignment that the middleware could practically support.

4.1 Patterns

We sketch some of the patterns here; these derive from those presented by Singh *et al.* [16]. Below, we describe a sample set of patterns that can readily be supported by the middleware.

Figure 4 shows the pattern for updating a commitment. At the programming level, this corresponds to the debtor sending an *Update* instruction to the middleware. At the computational level, the debtor's middleware sends two messages: one to cancel the existing commitment, and another to create a new commitment in its place.

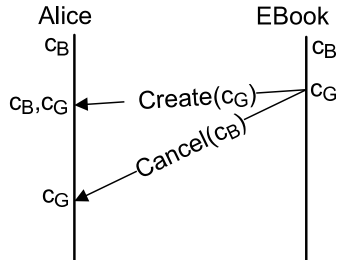


Fig. 4. Update pattern

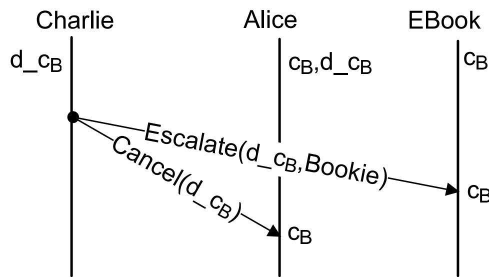


Fig. 5. Escalate pattern

Figure 5 shows the pattern for escalating a delegated commitment. The delegatee may find itself unable to fulfill the commitment. Here, the delegatee sends an *Escalate* instruction to the middleware. The middleware then sends a message notifying the delegator of the escalation of the commitment, and a $Cancel$ message to the creditor. (In the figures, a commitment with the name prefix $d_$ is the delegated version of a commitment. Since $c_B = \mathsf{C}(EBook, Alice, \$12, BNW)$, in Figure 5, $d_c_B = \mathsf{C}(Charlie, Alice, \$12, BNW)$.)

Figure 6 shows the pattern for delegating a commitment without retaining responsibility. Here, the debtor instructs the middleware to accomplish *DelegationWithoutResponsibility*. Along with the $Delegate$ instruction to the delegatee, the middleware sends a $Cancel$ message to the creditor thus absolving the debtor of any further responsibility.

Figure 7 shows the pattern for withdrawing a delegated commitment. The delegator sends a *Withdraw* instruction to the middleware. The middleware then sends a $Withdraw$ message to the delegatee. The delegatee's middleware, upon receiving this message, sends a $Cancel$ to the creditor. The callback for *Withdraw* would return in the delegatee.

Figure 8 shows the pattern for division of labor: different parts of the commitment are delegated to different parties. Here, the delivery of BNW is delegated to Charlie and that of GoW is delegated to Barnie.

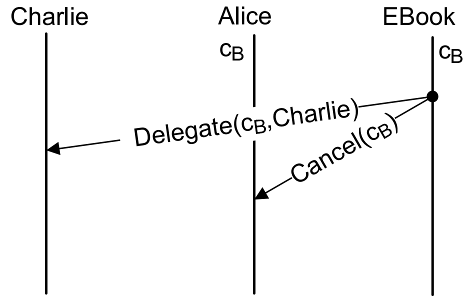


Fig. 6. Delegating without responsibility pattern

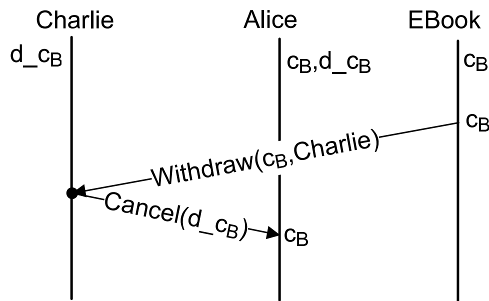


Fig. 7. Withdraw pattern

4.2 Other Forms of Alignment

Alignment as described in the previous sections and in [10] is essentially a creditor-debtor relation. When a creditor-debtor misalignment arises, there is the possibility of a violation of a commitment, and therefore, noncompliance. The following additional forms of alignment may be supported as additional patterns in the middleware. These forms of alignment may not necessarily result in noncompliance as it relates to commitment violation; nonetheless, these forms are useful for maintaining coherence in virtual organization settings, and are commonly effected in practice.

Debtor-debtor Alignment For example, suppose EBook delegates the commitment to send Alice a book to another bookseller Charlie. Then, EBook might want to be notified when Charlie discharges the commitment by sending the book, and vice versa.

Such alignment may be formalized in terms of debtor-debtor alignment: two agents who are related by a delegation relation remain aligned with respect to the discharge of the commitment. To effect such alignment would mean that the middleware would have to be configured with the additional constraint that if a debtor delegates a commitment to another agent, then whenever one of them discharges the commitment, it notifies the other.

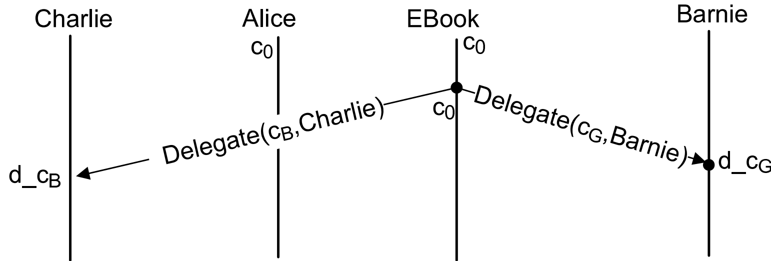


Fig. 8. Division of labor pattern

Considering alignment in a *debtor group* could also be a useful notion. When two or more agents are committed for the same thing (thus the group), then whenever one discharges the commitment, it notifies the entire group.

Creditor-creditor Alignment In a similar vein, suppose Alice assigns the commitment made to her by EBook to Bob. Alice may want to be notified when Bob sends the payment, and vice versa.

This alignment is between creditors, and it is formalized and effected analogously to debtor-debtor alignment.

Contextual Alignment Each commitment has a social or legal context. Although we have omitted the context from the commitment so far, each commitment is in general a relation between three agents, the debtor, the creditor, and the context, and is expressed as $\mathsf{C}(debtor, creditor, context, antecedent, consequent)$. The context's role is the enforcement of the commitment. If EBook and Alice are operating on eBay, then eBay is the context of their interaction. Applications such as eBay, in which the context itself plays an active role, typically have the requirement that the context should also be aligned with respect to the commitment.

Contextual alignment involves three parties; stronger guarantees, such as causal delivery [17] may be required from the communication layer.

5 Discussion: Conclusions and Future Work

In this paper, we have presented a multiagent system architecture based on interaction and commitments. In particular, we have introduced a middleware that can compute commitments and guarantee alignment between agents even in completely asynchronous settings. The middleware provides high-level programming abstractions that build on commitment operations.

Our architecture is unique in that commitments form the principal interconnections between agents. We deemphasize the implementation of the agent's engine. Agent programming languages, for example 2APL [18], remain largely based on BDI and do not support commitments. As mentioned before, such languages can be used to create

an agent's engine. Enhancing agent programming frameworks such as JADE with high-level abstractions, for example, as illustrated in [19], is no doubt useful. However, when one talks of multiagent systems, that invariably involves interaction and commitments. Therefore, a programming language or a framework for multiagent systems should ideally support reasoning about commitments, and have commitment-related abstractions. Even when interactions protocols are supported in agent-oriented methodologies and platforms, it is at the level of specific choreographies, and not of meaning (for example, [20–23]). Winikoff supports commitments in SAAPL by providing mappings from commitments to BDI-style plans, but the commitment reasoning supported is fairly limited [3].

The main priority for our research is the implementation of the proposed architecture. The language used here to give meanings to communications is sufficiently expressive for our purposes. We are investigating more powerful languages, however, for more subtle situations.

References

1. Yolum, P., Singh, M.P.: Flexible protocol specification and execution: Applying event calculus planning using commitments. In: Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems, ACM Press (July 2002) 527–534
2. Desai, N., Mallya, A.U., Chopra, A.K., Singh, M.P.: Interaction protocols as design abstractions for business processes. IEEE Transactions on Software Engineering **31**(12) (December 2005) 1015–1027
3. Winikoff, M.: Implementing commitment-based interactions. In: Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems. (2007) 1–8
4. Singh, M.P., Chopra, A.K., Desai, N.: Commitment-based SOA. IEEE Computer **42** (2009) Accepted; available from http://www.csc.ncsu.edu/faculty/mpsingh/papers/.
5. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Longman Publishing Co., Inc. (2003)
6. RosettaNet: Home page (1998) www.rosettanet.org.
7. http://www.gs1.org/productssolutions/gdsn/: GDSN
8. AMQP: Advanced message queuing protocol (2007) http://www.amqp.org.
9. Chopra, A.K., Singh, M.P.: Constitutive interoperability. In: Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems. (2008) 797–804
10. Chopra, A.K., Singh, M.P.: Multiagent commitment alignment. In: Proceedings of the 8th International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS), Columbia, SC, IFAAMAS (May 2009)
11. Singh, M.P.: An ontology for commitments in multiagent systems: Toward a unification of normative concepts. Artificial Intelligence and Law **7** (1999) 97–113
12. Fournet, C., Hoare, C.A.R., Rajamani, S.K., Rehof, J.: Stuck-free conformance. In: Proceedings of the 16th International Conference on Computer Aided Verification (CAV). Volume 3114 of LNCS., Springer (2004) 242–254
13. Bravetti, M., Zavattaro, G.: A theory for strong service compliance. In: Proceedings of 9th International Conference on Coordination Models and Languages (Coordination'07). Number 4467 in LNCS (2007) 96–112
14. Baldoni, M., Baroglio, C., Chopra, A.K., Desai, N., Patti, V., Singh, M.P.: Choice, interoperability, and conformance in interaction protocols and service choreographies. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems. (2009)

15. Singh, M.P.: Semantical considerations on dialectical and practical commitments. In: Proceedings of the 23rd Conference on Artificial Intelligence. (2008) 176–181
16. Singh, M.P., Chopra, A.K., Desai, N.: Commitment-based SOA. IEEE Computer **42** (2009) Accepted; available from http://www.csc.ncsu.edu/faculty/mpsingh/papers/.
17. Schiper, A., Birman, K., Stephenson, P.: Lightweight causal and atomic group multicast. ACM Transactions on Computer Systems **9**(3) (1991) 272–314
18. Dastani, M.: 2APL: A practical agent programming language. Autonomous Agents and Multi-Agent Systems **16**(3) (20078) 214–248
19. Baldoni, M., Boella, G., Genovese, V., Grenna, R., van der Torre, L.: How to program organizations and roles in the jade framework. In: Multiagent System Technologies. Volume 5244 of LNCS., Springer (2008) 25–36
20. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Developing multiagent systems: The Gaia methodology. ACM Transactions on Software Engineering Methodology **12**(3) (2003) 317–370
21. Padgham, L., Winikoff, M.: Prometheus: A practical agent-oriented methodology. In Henderson-Sellers, B., Giorgini, P., eds.: Agent-Oriented Methodologies. Idea Group, Hershey, PA (2005) 107–135
22. Garcia-Ojeda, J.C., DeLoach, S.A., Robby, Oyenan, W.H., Valenzuela, J.: O-MaSE: A customizable approach to developing multiagent processes. In: Proceedings of the 8th International Workshop on Agent Oriented Software Engineering (AOSE 2007). (2007)
23. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: The sciff framework. ACM Transactions on Computational Logic **9**(4) (2008)

A Computational Semantics for Communicating Rational Agents Based on Mental Models

Koen V. Hindriks and M. Birna van Riemsdijk

EEMCS, Delft University of Technology, Delft, The Netherlands
{k.v.hindriks,m.b.vanriemsdijk}@tudelft.nl

Abstract. Communication is key in a multi-agent system for agents to exchange information and coordinate their activities. For agents that derive their choice of action from their beliefs and goals it is natural to facilitate communication about both these attitudes in an agent programming language. The traditional approach based on speech act theory, however, does not provide the right tools to do so because of its emphasis on mental conditions on the speaker. Here, we introduce an alternative semantics based on the idea that a received message can be used to *(re)construct a mental model* of the sender. As coordination is particularly important, we introduce the concept of a *conversation* to synchronize actions and communication in a multi-agent system. Conversations are resources at the multi-agent level with restricted access, which provide a natural counterpart in multi-agent systems for classic constructs from distributed programming such as semaphores.

1 Introduction

Communication is key in a multi-agent system for agents to exchange information and coordinate their activities, and therefore needs to be addressed in agent programming languages. Two main lines of research can be identified when it comes to agent communication languages: research based on speech act theory [1, 2] and research based on a social semantics for agent communication (social commitments) [3, 4].

Speech act theory is a philosophical theory that is based on the idea that uttering a sentence is an act which can be used to change the world like any other act. The focus of speech act theory has been on specifying the conditions that identify the particular act that is performed, thereby focussing most of the theory on the sender instead of the receiver. Two of the most well-known agent communication languages that are based on speech act theory are KQML [5] and the FIPA Agent Communication Language (ACL) [6]. Both languages specify the semantics of messages by means of their pre-condition, expressing conditions on the mental states of the sender and receiver of the message that should hold if the message is sent, and their effect, expressing the effect of the message on the mental state of the sending and/or receiving agent.[1] Both languages use *performative labels* to specify message types. For example, the precondition for the

[1] FIPA does not specify that there should be any effect on the sender [6].

(simplified) FIPA *inform*(r, ϕ) message, where *inform* denotes the performative label and ϕ the content of the message, includes that the sending agent believes ϕ and the effect is that the receiver r believes ϕ.

These ACLs have been extensively criticized [7–9]. Here, we mention in particular their *complexity* due to the relatively high number of performative labels and subtle semantical differences between them, and the lack of *verifiability*. The latter means that, for example, an agent receiving a message that is supposed to be an informative speech act with preconditions that require truth of the content, cannot verify these conditions. An important alternative approach that addresses these and other issues, is the semantics for agent communication based on social commitments [3, 4]. This approach does not define communication by referring to the mental states of the involved agents, but focuses on the social consequences of communication. The basic idea here is that a receiver can always confront the sender again with a previous message, i.e., by saying something like: "You told/asked/requested me so". The processing of the message by the receiver, however, has moved to the background here.

We can thus see that communication based on speech act theory is problematic due to its complexity and verifiability. Moreover, the approach based on social commitments does not say anything about how communication affects the involved agents. This issue *does* need to be addressed when developing techniques for communication in agent programming languages, which is what we are interested in in this paper. The contribution of this paper is the introduction of an alternative semantics based on the idea that a received message can be used to *(re)construct a mental model* of the sender. Our semantics does not specify any preconditions on the mental states of the agents for sending a message. Moreover, the effect on the receiver is only that it updates its mental model of the sender, i.e., the mental state of the receiver itself is not directly updated. A second contribution is the introduction of the concept of a *conversation* to facilitate the synchronization of actions and communication in a multi-agent system, which is particularly important to organize agent coordination. Our proposal is made concrete in the agent programming language GOAL [10].

2 Communication in Agent Programming Languages

Communication within other agent programming languages than GOAL has taken a quite pragmatic turn to address the issue discussed above. For example, in 2APL the semantics of communication is reduced to a simple "mailbox" semantics: communicating a message only means that the message is added to a mailbox and the programmer then needs to write rules to handle received messages [11]. This reduces the meaning of communication to the bare minimum. The developers of Jason have chosen a small set of primitives, based on KQML, for which simple default semantics have been defined [8]. For example, the *tell* message inserts the content of the message into the receiver's belief base, with a tag to identify the source of this information, and the *achieve* message inserts the content of the message as a goal in the event base. A function is

introduced to determine whether a message is "socially acceptable", and only socially acceptable messages are processed.

Although we take a definite engineering stance in this paper regarding the design of a semantics for communication, our approach is intended to satisfy a number of basic criteria. First, the communication primitives should have a well-defined *semantics*. Preferably, this semantics provides a basis for verifying multi-agent systems as well, where agents use these primitives to communicate. Second, *the distinction between beliefs and goals* needs to be taken into account when defining a communication semantics. As GOAL agents derive their choice of action from their beliefs and goals, it is important to be able to communicate about these different reasons for acting as well as to be able to distinguish between beliefs and goals communicated by other agents. Third, the communication primitives introduced should be *useful for programming* GOAL *agents*. This is a pragmatic criterion that requires minimization of complexity, in particular by limiting the number of communication primitives and endowing these with easy to grasp semantics. Fourth, and finally, *various speech acts should be definable* using the communication primitives introduced. Ideally, it would be possible to define or "program" various well-known speech acts such as promises, requests, etc. in terms of the primitives introduced. It should be kept in mind, however, that characterizing a particular communication event of a multi-agent system may be possible only from a designer or observer's point of view, e.g. by using a *logic* to *reason about* the communication primitives. Initial work to meet this last criterium, which sets our approach apart from others, is reported in [12].

3 Redesigning Agent Communication

Our proposal for redesigning agent communication in the context of the programming language GOAL focuses on the effects of communication on the receiver, a perspective also taken in [13, 8] but which may be contrasted with social commitment semantics. Doing so is not straightforward and requires that some reasonable decisions are made with respect to defining a communication semantics. We illustrate this using an example (see also [14]). Consider the utterance "The house is white". Its effect on the receiver may be one or more of the following, ranging from a very strong to a very weak effect:

1. The receiver comes to believe that the house is white.
2. The receiver comes to believe that the sender believes that the house is white.
3. The receiver comes to believe that the sender had the intention to make the receiver believe that the house is white.
4. The utterance has no effect on the receiver, i.e. its mental state is not changed as a result of the utterance.

In choosing which of these effects should form the basis for our semantics, we take a pragmatic engineering stance, and in addition want to avoid making too strong assumptions. In particular, we consider effect 1 to be too strong in general, as it makes the assumption that the sender always convinces the receiver.

Effect 4 is too weak, and not very useful for programming communication among agents since it would have no effect. Effect 3 would be rather indirect, as it is no longer very clear what *use* the communication has, other than to conclude such indirect statements about the sender's mind. In practice, using this type of semantics would be rather similar to effect 4, doing nothing with a message received from another agent, as making good use of information about the intentions of another agent would require rather involved reasoning patterns. Here, we choose our semantics according to effect 2, which means that the receiver makes the assumption that the speaker believes what it says. Obviously, this is not always a safe assumption to make, as the sender may be lying. However, it is also not overly presumptuous, as the receiver just takes the utterance of the sender at face value.

In contrast with effect 1, effect 2 does not affect the mental state of the receiver directly, but only the *mental model* that the receiver has of the sender: as a result of the utterance, the receiver comes to believe something about the sender, but the utterance does not directly influence the beliefs of the agent about the environment. This is one of the main ideas of our approach: the direct effect of communication is that the receiver updates its mental model of the sender. Additional reasoning may then result in the receiver making updates also to its own beliefs and goals. However, this is not part of our semantics. For example, if the receiver knows by experience that the sender is quite reliable, the receiver may also update its own beliefs accordingly, or use the beliefs of the mental model of the sender to decide on action (as in the example program of Section 5). This is in contrast with Jason, in which reasoning about the acceptability of the message is done upon receipt of the message (using the function to determine the social acceptability), and if the message is acceptable, the mental state of the agent itself is updated.

The second main idea behind our approach has to do with the types of messages that we distinguish. For this, we take inspiration from natural language, in which one uses grammatical structure to differentiate between various types of communication modes. In the communication framework we propose, we distinguish between *three message types*, derived from three *grammatical* distinctions in natural language:

(i) *declaratives*, typically used to make factual statements about the environment (e.g., "The house is white."). Syntactically, a declarative is represented by $\bullet\phi$. Informally, a declarative message with content ϕ may be paraphrased as: "It is the case that ϕ." Semantically, the idea is that the receiver r takes this at face value, and r concludes that sender s believes ϕ.

(ii) *interrogatives*, typically used to pose questions about a state of affairs (e.g., "Is the house white?"). Syntactically, an interrogative is represented by $\mathbf{?}\phi$. Informally, an interrogative with content ϕ may be paraphrased as: "Is it the case that ϕ?". Taking this at face value, r concludes that s does not know whether ϕ.

(iii) *imperatives*, typically used to express a desirable state of affairs (e.g., "See to it that the house is white!"). Syntactically, an imperative is represented

by $\mathbf{!}\phi$. Informally, an imperative may be paraphrased as: "Someone, see to it that ϕ." Taking this at face value, r concludes that s has ϕ as a goal, and does not believe ϕ.[2]

The semantics of interrogatives and imperatives have in common with that of declaratives that they do not prescribe what the receiver should do in terms of updating its own beliefs and goals. The semantics of interrogatives does not define that the receiver should, e.g., adopt the goal to tell the sender about ϕ. Similarly, the semantics of imperatives does not define that the receiver should update its own goals with ϕ. Moreover, the semantics of imperatives does not even define whether the uttered imperative should be interpreted as a request, or simply as information about the goals of the sender. Adding these kinds of interpretations and additional reasoning on whether to update the receiver's own beliefs and goals is *left to the agent programmer*. For example, in certain applications imperatives might always be interpreted as requests, while in others one might want to make a distinction between these and imperatives that express the goals of the sender. However, we argue that in all of these cases, it makes sense to update the mental model that the receiver has of the sender as informally described above.

4 A Communication Semantics Based on Mental Models

In this section, we make the informal semantics discussed above precise in the context of GOAL.

4.1 Mental Models and Mental States

Mental models play an essential role in this semantics and are introduced first. GOAL agents maintain mental models that consists of *declarative* beliefs and goals. An agent's beliefs represent its environment whereas the goals represent a state of the environment the agent wants. Beliefs and goals are specified using some knowledge representation technology. In the specification of the operational semantics we use a propositional logic $\mathcal{L}_0$ built from a set of propositional atoms *Atom* and the usual boolean connectives. We use $\models$ to denote the usual consequence relation associated with $\mathcal{L}_0$, and assume a special symbol $\bot \in \mathcal{L}_0$ which denotes the false proposition. In addition, the presence of an operator $\oplus$ for adding ϕ to a belief base and an operator $\ominus$ for removing ϕ from a belief base are assumed to be available.[3] A mental model associated with a GOAL agent needs to satisfy a number of *rationality constraints*.

[2] The latter part, that s does not believe ϕ is derived from a rationality constraint. An agent should not have a goal to achieve something if it believes it has already been achieved.

[3] We assume that $\Sigma \oplus \phi \models \phi$ whenever ϕ is consistent, and that otherwise nothing changes, and that $\Sigma \ominus \phi \not\models \phi$ whenever ϕ is not a tautology, and that otherwise nothing changes. Additional properties such as minimal change, etc. are usually associated with these operators (see e.g. [15]) but not relevant in this context.

Definition 1. *(Mental Model)*
A mental model is a pair $\langle \Sigma, \Gamma \rangle$ *with* $\Sigma, \Gamma \subseteq \mathcal{L}_0$ *such that:*

- *The beliefs are consistent:* $\Sigma \not\models \bot$
- *Individual goals are consistent:* $\forall \gamma \in \Gamma : \gamma \not\models \bot$
- *Goals are not yet (believed to be) achieved:* $\forall \gamma \in \Gamma : \Sigma \not\models \gamma$

In a multi-agent system it is useful for an agent to maintain mental models of other agents. This allows an agent to keep track of the perspectives of other agents on the environment and the goals they have adopted to change it. A mental model maintained by an agent i about another agent j represents what i thinks that j believes and which goals it has. Mental models of other agents can also be used to take the beliefs and goals of these agents into account in its own decision-making. An agent may construct a mental model of another agent from the messages it receives from that agent or from observations of the actions that that agent performs (e.g., using intention recognition techniques). Here we focus on the former option.

We assume a multi-agent system that consists of a fixed number of agents. To simplify the presentation further, we use $\{1, \ldots, n\}$ as names for these agents. A *mental state* of an agent is then defined as a mapping from all agent names to mental models.

Definition 2. *(Mental State)*
A mental state m *is a total mapping from agent names to mental models, i.e.* $m(i) = \langle \Sigma_i, \Gamma_i \rangle$ *for* $i \in \{1, \ldots, n\}$.

For an agent i, $m(i)$ are its own beliefs and goals, which was called the agent's mental state in [10].

A Goal agent is able to inspect its mental state by means of *mental state conditions*. The mental state conditions of Goal consist of atoms of the form $\mathbf{bel}(i, \phi)$ and $\mathbf{goal}(i, \phi)$ and Boolean combinations of such atoms. $\mathbf{bel}(i, \phi)$ where i refers to the agent itself means that the agent itself believes ϕ, whereas $\mathbf{bel}(i, \phi)$ where i refers to another agent means that the agent believes that agent i believes ϕ. Similarly, $\mathbf{goal}(i, \phi)$ is used to check whether agent i has a goal ϕ.[4]

Definition 3. *(Syntax of Mental State Conditions)*
A mental state condition*, denoted by* ψ*, is defined by the following rules:*

$$
\begin{aligned}
i &::= \textit{any element from } \{1, \ldots, n\} \mid \textbf{\textit{me}} \mid \textbf{\textit{allother}} \\
\phi &::= \textit{any element from } \mathcal{L}_0 \\
\psi &::= \mathbf{bel}(i, \phi) \mid \mathbf{goal}(i, \phi) \mid \psi \wedge \psi \mid \neg \psi
\end{aligned}
$$

The meaning of a mental state condition is defined by means of the mental state of an agent. An atom $\mathbf{bel}(i, \phi)$ is true whenever ϕ follows from the belief

[4] In a multi-agent setting it is useful to introduce additional labels instead of agent names i, e.g. **me** to refer to the agent itself and **allother** to refer to all other agents, but we will not discuss these here in any detail.

base of the mental model for agent i. An atom $\mathbf{goal}(i, \phi)$ is true whenever ϕ follows from *one* of the goals of the mental model for agent i. This is in line with the usual semantics for goals in GOAL, which allows the goal base to be inconsistent (see [10] for details). Note that we overload $\models$.

Definition 4. *(Semantics of Mental State Conditions)*
Let m be a mental state and $m(i) = \langle \Sigma_i, \Gamma_i \rangle$. Then the semantics of mental state conditions is defined by:

$$\begin{array}{lll} m \models \mathbf{bel}(i,\phi) & \textit{iff} & \Sigma_i \models \phi \\ m \models \mathbf{goal}(i,\phi) & \textit{iff} & \exists \gamma \in \Gamma_i \textit{ such that } \gamma \models \phi \\ m \models \neg\psi & \textit{iff} & m \not\models \psi \\ m \models \psi \wedge \psi' & \textit{iff} & m \models \psi \textit{ and } m \models \psi' \end{array}$$

4.2 Actions

GOAL has a number of built-in actions and also allows programmers to introduce user-specified actions by means of STRIPS-style action specifications. The program discussed in Section 5 provides examples of various user-specified actions. In the definition of the semantics we will abstract from action specifications specified by programmers and assume that a fixed set of actions *Act* and a (partial) transition function T is given. T specifies how actions from *Act*, performed by agent i, update i's mental state, i.e., $T(i, a, m) = m'$ for i an agent name, $a \in Act$ and m, m' mental states. All actions except for communicative actions are assumed to only affect the mental state of the agent performing the action.

The built-in actions available in GOAL (adapted to distinguish between mental models) that we need here include $\mathbf{ins}(i, \phi)$, $\mathbf{del}(i, \phi)$, $\mathbf{adopt}(i, \phi)$, $\mathbf{drop}(i, \phi)$ and communicative actions of the form $\mathbf{send}(i, msg)$ where i is an agent name and msg is a message of the form $\bullet\phi$, $\mathbf{?}\phi$ or $\mathbf{!}\phi$. The semantics of actions from *Act* and built-in actions performed by agent i is formally captured by a mental state transformer function M defined as follows:

$$\begin{array}{lll} M(i,a,m) & = & \begin{cases} T(i,a,m) & \text{if } a \in Act \text{ and } T(i,a,m) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases} \\ M(i,\mathbf{ins}(j,\phi),m) & = & m \oplus_j \phi \\ M(i,\mathbf{del}(j,\phi),m) & = & m \ominus_j \phi \\ M(i,\mathbf{adopt}(j,\phi),m) & = & \begin{cases} m \cup_j \phi & \text{if } \phi \text{ is consistent and } m \not\models \mathbf{bel}(i,\phi) \\ \text{undefined} & \text{otherwise} \end{cases} \\ M(i,\mathbf{drop}(j,\phi),m) & = & m -_j \phi \\ M(i,\mathbf{send}(j,msg),m) & = & m \end{array}$$

where $m \times_j \phi$ means that operator $\times \in \{\oplus, \ominus, \cup, -\}$ is applied to mental model $m(j)$, i.e. $m \times_j \phi(i) = m(j) \times \phi$ and $m \times_j \phi(k) = m(k)$ for $k \neq j$. To define the application of operators to mental models, we use $Th(T)$ to denote the logical theory induced by T, i.e. the set of all logical consequences that can be derived from T. Assuming that $m(i) = \langle \Sigma, \Gamma \rangle$, we then define: $m(i) \oplus \phi = \langle \Sigma \oplus \phi, \Gamma \setminus (Th(\Sigma \oplus \phi) \rangle$, $m(i) \ominus \phi = \langle \Sigma \ominus \phi, \Gamma \rangle$, $m(i) \cup \phi = \langle \Sigma, \Gamma \cup \{\phi\} \rangle$, and

$m(i) - \phi = \langle \Sigma, \Gamma \setminus \{\gamma \in \Gamma \mid \gamma \models \phi\}\rangle$. Note that sending a message does not have any effect on the sender. There is no need to incorporate any such effects in the semantics of **send** since such effects may be *programmed* by using the other built-in operators.

It is useful to be able to perform multiple actions simultaneously and we introduce the + operator to do so. The idea here is that multiple mental actions may be performed simultaneously, possibly in combination with the execution of a *single* user-specified action (as such actions may have effects on the external environment it is not allowed to combine multiple user-specified actions by the + operator). The meaning of $a + a'$ where a, a' are actions, is defined as follows: if $M(i, a, m)$ and $M(i, a', m)$ are defined and $M(i, a', M(i, a, m)) = M(i, a, M(i, a', m))$ is a mental state, then $M(i, a{+}a', m) = M(i, a', M(i, a, m))$; otherwise, $a + a'$ is undefined.

In order to select actions for execution, an agent uses action rules of the form **if** ψ **then** a, where a is a user-specified action, a built-in action, or a combination using the +-operator. An agent $\mathcal{A}$ is then a triple $\langle i, m, \Pi\rangle$ where i is the agent's name, m is the agent's mental state, and Π is the agent's program (a set of action rules).

4.3 Operational Semantics: Basic Communication

We first introduce a single transition rule for an agent performing an action. Transitions "at the agent level" are labelled with the performed action, since this information is required "at the multi-agent level" in the case of communicative actions.

Definition 5. (Actions)
Let $\mathcal{A} = \langle i, m, \Pi\rangle$ be an agent, and **if** ψ **then** $\mathsf{a} \in \Pi$ *be an action rule.*

$$\frac{m \models \psi \quad M(i, \mathsf{a}, m) \textit{ is defined}}{m \xrightarrow{\mathsf{a}} M(i, \mathsf{a}, m)}$$

Using Plotkin-style operational semantics, the semantics at the multi-agent level is provided by the rules below. A configuration of a multi-agent system consists of the agents of the multi-agent system $\{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ and the environment E, which is used to store messages that have been sent and are waiting for delivery.[5] The environment is used to model *asynchronous* communication, i.e., no handshake is required between sender and receiver of a message. Transitions at the multi-agent level are not labelled. Actions other than the **send** action only change the agent that executes them, as specified below.

Definition 6. (Action Execution)
Let "a" be an action other than send(j, msg).

$$\frac{\mathcal{A}_i \xrightarrow{a} \mathcal{A}_i'}{\mathcal{A}_1, \ldots, \mathcal{A}_i, \ldots, \mathcal{A}_n, E \longrightarrow \mathcal{A}_1, \ldots, \mathcal{A}_i', \ldots, \mathcal{A}_n, E}$$

[5] Other aspects of the environment might also be modeled, but that is beyond the scope of this paper.

The following transition rule specifies the semantics of sending messages.

Definition 7. (Send)

$$\frac{\mathcal{A}_i \overset{send(j,msg)}{\longrightarrow} \mathcal{A}_i}{\mathcal{A}_1,\ldots,\mathcal{A}_i,\ldots,\mathcal{A}_n,E \longrightarrow \mathcal{A}_1,\ldots,\mathcal{A}_i,\ldots,\mathcal{A}_n,E \cup \{send(i,j,msg)\}}$$

The premise of the rule indicates that agent $\mathcal{A}_i$ sends a message to agent $\mathcal{A}_j$. To record this, $send(i,j,msg)$ is added to the environment, including both the sender i and the intended receiver j. Also note that a message that is sent more than once has no effect as the environment is modeled as a set here (this is the case until the message is received).[6]

Three rules for receiving a message are introduced below, corresponding to each of the three message types. In each of these rules, the conclusion of the rule indicates that the mental state of the receiving agent is changed. If agent j receives a message from agent i that consists of a declarative sentence, it has the effect that the mental model $m(i)$ of the mental state of the receiver j is modified by updating the belief base of $m(i)$ with ϕ. In addition, any goals in the goal base of $m(i)$ that are implied by the updated belief base are removed from the goal base to ensure that the rationality constraints associated with mental models are satisfied.

Definition 8. (Receive: Declaratives)

$$\frac{send(i,j,\bullet\phi) \in E}{\mathcal{A}_1,\ldots,\langle j,m,\Pi\rangle,\ldots,\mathcal{A}_n,E \longrightarrow \mathcal{A}_1,\ldots,\langle j,m',\Pi\rangle,\ldots,\mathcal{A}_n,E \setminus \{send(i,j,\bullet\phi)\}}$$

where:

- $m'(i) = \langle \Sigma \oplus \phi, \Gamma \setminus Th(\Sigma \oplus \phi)\rangle$ *if* $m(i) = \langle \Sigma, \Gamma\rangle$, *and*
- $m'(k) = m(k)$ *for* $k \neq i$.

The condition $m'(k) = m(k)$ for $k \neq i$ ensures that only the mental model associated with the sender i is changed.

The rule below for interrogatives formalizes that if agent i communicates a message $\mathbf{?}\varphi$ of the interrogative type, then the receiver j will assume that i does not know the truth value of ϕ. Accordingly, it removes ϕ using the $\ominus$ operator from the belief base in its mental model of agent i to reflect this.

Definition 9. (Receive: Interrogatives)

$$\frac{send(i,j,\,?\phi) \in E}{\mathcal{A}_1,\ldots,\langle j,m,\Pi\rangle,\ldots,\mathcal{A}_n,E \longrightarrow \mathcal{A}_1,\ldots,\langle j,m',\Pi\rangle,\ldots,\mathcal{A}_n,E \setminus \{send(i,j,\,?\phi)\}}$$

where:

- $m'(i) = \langle (\Sigma \ominus \phi)) \ominus \neg\phi, \Gamma\rangle$ *if* $m(i) = \langle \Sigma, \Gamma\rangle$, *and*
- $m'(k) = m(k)$ *for* $k \neq i$.

[6] The implicit quantifier **allother** may be used to define a broadcasting primitive: $\mathbf{broadcast}(msg) \overset{df}{=} \mathbf{send}(\mathbf{allother}, msg)$. In the rule above, in that case, for all $i \neq j$ $send(i,j,msg)$ should be added to E, but we do not provide the details here.

Remark An alternative, more complex semantics would not just conclude that agent i does not know ϕ but also that i wants to know the truth value of ϕ, introducing a complex proposition $K_i\phi$ into the model of the goal base of that agent. This would involve nesting of operators, or including $K_i\phi$ in the goal base.

The rule below for imperatives formalizes that if agent i communicates a message $!\phi$ of the imperative type, then the receiver j will assume that i does not believe that ϕ is the case, and also that ϕ is a goal of i. Accordingly, it removes ϕ using the $\ominus$ operator and adds ϕ to its model of the goal base of agent i.

Definition 10. (Receive: Imperatives)

$$\frac{send(i, j, ?\phi) \in E}{\mathcal{A}_1, \ldots, \langle j, m, \Pi\rangle, \ldots, \mathcal{A}_n, E \longrightarrow \mathcal{A}_1, \ldots, \langle j, m', \Pi\rangle, \ldots, \mathcal{A}_n, E \setminus \{send(i, j, ?\phi)\}}$$

where:

- $m'(i) = \langle \Sigma \ominus \phi, \Gamma \cup \{\phi\}\rangle$ *if* $\phi \not\models \bot$ *and* $m(i) = \langle \Sigma, \Gamma\rangle$; *otherwise,* $m'(i) = m(i)$.
- $m'(k) = m(k)$ *for* $k \neq i$.

Note that this semantics does not refer to the *actual* mental state of the sender, nor does it define when a sender should send a message or what a receiver should do with the contents of a received message (other than simply record it in its mental model of the sending agent).

4.4 Operational Semantics: Conversations

As is well-known, in concurrent systems one needs mechanisms to ensure that processes cannot access a particular resource simultaneously. A similar need arises in multi-agent systems, but this has received little attention in the agent programming community so far. Emphasis has been put on the fact that agent communication is *asynchronous*. However, in order to ensure that only one agent has access to a particular resource at any time, agents need to be able to coordinate their activities and *synchronize* their actions.[7] Of course, asynchronous communication allows to implement synchronization between agents. We argue, however, that it is useful to have predefined primitives available in an agent programming language that facilitate coordination and synchronization, as is usual in concurrent programming [16]. We introduce a mechanism that fits elegantly into the overall setup of communication primitives introduced above, using the notion of a *conversation*.

The basic idea is that an agent can engage only in a limited number of conversations at the same time. By viewing a conversation as a resource, the

[7] Note that *perfectly symmetrical solutions to problems in concurrent programming are impossible because if every process executes exactly the same program, they can never 'break ties'* [16]. To resolve this, solutions in concurrency theory contain asymmetries in the form of process identifiers or a kernel maintaining a queue.

limit on the number of conversations an agent can participate in simultaneously thus introduces a limit on access to that resource. For our purposes, it will suffice to assume that an agent can participate in at most one conversation at the same time.

More specifically, a parameter representing a unique conversation identifier can be added when sending a message, i.e., **send**$(c : j, msg)$ specifies that the message msg should be sent to agent j as part of the ongoing conversation c. We also allow conversations with groups of more than two agents which is facilitated by allowing groups of agent names $\{\ldots\}$ to be inserted into **send**$(c : \{\ldots\}, msg)$. A message that is sent as part of an ongoing conversation c is handled similarly to a message that is not part of a specific conversation. Whenever a conversation c has been closed (see below), sent messages that are intended to be part of that conversation are "lost", i.e. nothing happens. To initiate a conversation, the term **new** can be used instead of the conversation identifier. That is, whenever an agent i performs a **send**(**new** $: g, msg)$ action where g is an agent or a group of agents, agent i initiates a new conversation. Because agents can only engage in a limited number of conversations at the time, it may be that an initiated conversation is *put on hold initially* because one of the agents that should participate already participates in another conversation.

Semantically, to be able to model that a conversation is ongoing, we split the environment into a set A of *active conversations*, a queue Q of *pending conversations*, and a set M of other pending messages. A message to initiate a new conversation is added to the queue *if* at least one agent that should participate is already present in the set A or the queue Q. The check on Q guarantees that a conversation is not started when another conversation requiring the participation of one of the same agents is still on hold in the queue ("no overtaking takes place"). Otherwise, the message is directly added to the set of active conversations.

Whenever a message $send(c : i, g, msg)$ that initiated a conversation is part of the set A, written $c \in A$, we will say that *conversation c is ongoing*, and when such a message is part of the queue Q, written $c \in Q$, we will say that *conversation c is put on hold*. Since the rules for receiving messages remain essentially the same, we only provide the rules for sending a message at the multi-agent level. The following rule specifies the semantics of sending a message that is part of an ongoing conversation.

Definition 11. (Send: Ongoing Conversation)

$$\frac{\mathcal{A}_i \overset{send(c:j,msg)}{\longrightarrow} \mathcal{A}'_i \quad c \in A}{\mathcal{A}_1, \ldots, \mathcal{A}_i, \ldots, \mathcal{A}_n, \langle A, Q, M \rangle \longrightarrow \mathcal{A}_1, \ldots, \mathcal{A}'_i, \ldots, \mathcal{A}_n, \langle A, Q, M' \rangle}$$

where $M' = M \cup \{send(c : i, j, msg)\}$.

The following transition rule specifies the semantics of messages that are used to initiate conversations. We use $+$ (e.g., $Q{+}send(c : i, g, msg)$) to add a message to the tail of a queue. The set of active conversations A and the queue Q store

information about participants in conversations, as this may be derived from $\mathbf{send}(c : i, g, msg)$, where agents i and g are participants. We write $agent(A, Q)$ to denote the set of agents in A and Q.

Definition 12. (Send: Initiating a Conversation)
Let g be a set of agent names, and c a new conversation identifier not yet present in A or Q.

$$\frac{\mathcal{A}_i \stackrel{send(\mathbf{new}:g,msg)}{\longrightarrow} \mathcal{A}'_i}{\mathcal{A}_1, \ldots, \mathcal{A}_i, \ldots, \mathcal{A}_n, \langle A, Q, M\rangle \longrightarrow \mathcal{A}_1, \ldots, \mathcal{A}'_i, \ldots, \mathcal{A}_n, \langle A', Q', M'\rangle}$$

where if $(\{i\} \cup g) \cap agents(A, Q) = \emptyset$ then $A' = A \cup \{send(c : i, g, msg)\}$, $Q' = Q$ and $M' = \bigcup_{k \in g} send(c : i, k, msg)$, and otherwise $A' = A$, $Q' = Q + send(c : i, g, msg)$, and $M' = M$.

This semantics specifies that we cannot simply allow a conversation between two agents to start when these agents are not part of an ongoing conversation, as this may prevent a conversation between another group of agents involving the same agents from ever taking place. The point is that it should be prevented that "smaller" conversations always "overtake" a conversation between a larger group of agents that is waiting in the queue.

As conversations are a resource shared at the multi-agent level, it must be possible to free this resource again. To this end, we introduce a special action $\mathbf{close}(c)$ which has the effect of removing an ongoing conversation from the set A and potentially adding conversations on hold from the queue Q to A. This is the only essentially new primitive needed to implement the conversation synchronization mechanism.

We need an additional definition: we say that F is a *maximal fifo-set of messages* derived from a queue Q relative to a set of agent names Agt if F consists of *all* messages $send(c : i, g, msg)$ from Q that satisfy the following constraints: (i) $(\{i\} \cup g) \cap Agt = \emptyset$, and (ii) there is no earlier message $send(c' : i', g', msg')$ in the queue Q such that $(\{i\} \cup g) \cap g' \neq \emptyset$.

Definition 13. (Send: Closing a Conversation)

$$\frac{\mathcal{A}_i \stackrel{close(c)}{\longrightarrow} \mathcal{A}'_i}{\mathcal{A}_1, \ldots, \mathcal{A}_i, \ldots, \mathcal{A}_n, \langle A, Q, M\rangle \longrightarrow \mathcal{A}_1, \ldots, \mathcal{A}'_i, \ldots, \mathcal{A}_n, \langle A', Q', M\rangle}$$

where, assuming that F is the maximal fifo-set derived from Q relative to $agents(A)$, if $send(c : i, g, msg) \in A$ then $A' = (A \setminus \{send(c : i, g, msg)\}) \cup F$ and $Q' = Q \setminus F$, and otherwise $A' = A$ and $Q' = Q$.

Note that the transition rule for closing a conversation only allows the initiator of a conversation, i.e. agent $\mathcal{A}_i$, to close the conversation again. (Otherwise agents that want to start their own conversation immediately might try to get it going by closing other conversations.) Finally, as it is important that the initiating agent as well as other participating agents are aware that a conversation

has started or is ongoing, we assume a special predicate $conversation(c, i)$ is available, where c denotes a unique conversation identifier and i the initiating agent, which can be used in the belief base of an agent to verify whether a conversation is ongoing or not. We do not provide the formal details here due to space restrictions (see the next section for an example).

5 The Dining Philosophers

The dining philosophers is a classic problem in concurrency theory [16]. Below, we show parts of a Goal program that implements a solution. The complete program (for one philosopher agent) is listed in Appendix A. The currently implemented version of Goal uses Prolog as a knowledge representation language, which we also use here. We use numbers to refer to the action rules of the Goal program. For convenience, when referring to the agent's own mental model in mental state conditions, we drop this parameter.

A number of philosophers are sitting at a round table where they each engage in two activities: thinking and eating (1,2). Our philosophers only think when they are not hungry and get hungry after thinking a while (see the action specifications). At the table an unlimited supply of spaghetti is available for eating. A philosopher needs two forks, however, to be able to eat (3). Forks are available as well, but the number of forks equals the number of philosophers sitting at the table (one fork is between each two philosophers). It is thus is never possible for all of the philosophers to eat at the same time and they have to coordinate. The problem is how to ensure that each philosopher will eventually be able to eat.

Using our conversational metaphor for coordinating activities, the problem of the dining philosophers can be solved elegantly at the knowledge level. In the solution we present, the dining philosophers are assumed to be decent agents that are always willing to listen to the needs of their fellow philosophers at the table, and provide them with the forks when they indicate they require the forks to eat. If a philosopher needs the forks to eat but they are not available, he will initiate a conversation with his neighbors and indicate that he needs the forks (4).[8] If a philosopher i is eating and receives a request for forks from a fellow philosopher X as part of a new conversation, i will finish eating and put down the fork in between X and himself and notify X of this fact (8). A philosopher i will put down a fork only upon being requested. As long as the conversation is ongoing, i will not pick up the fork again. The philosopher that initiated the conversation will pick up the fork after being informed by his neighbor that the fork is on the table (6).[9] The initiator of the conversation informs his neighbors that he picked up the fork (6). Upon receiving a message from both neighbors

[8] In the sent messages the direction of the forks (left, right) has been dropped as this is just a matter of perspective, useful for keeping track of which fork has been picked up or put down from a single philosopher's perspective. From the point of view of two philosophers, a fork is just "in between" them.

[9] In fact, only when having initiated a conversation to require the forks, will a philosopher pick up a fork in our solution.

that they do not know whether the fork is on the table or not (reflected in the mental models of the neighbors), the initiator closes the conversation (7), and another conversation involving one of the philosophers may be started. Rules 5 and 9 are used to update the philosopher's own beliefs on the basis of its mental models of other philosophers (which are changed due to the sending of messages).

```
% i is the name of this philosopher agent
beliefs{ hold(fork,left). }  goals{ hold(fork,left), hold(fork,right). }
program{
 1. if true then think.
 2. if true then eat.
 3. if bel(hungry) then adopt(hold(fork,left), hold(fork,right)).
 4. if goal(hold(fork,_)), bel(not(forksAvailable), neighbours(X,Y))
   then send(new:{X,Y},!hold(fork)).
 5. if bel(neighbour(X,D), not(hold(fork,D))), bel(X, on(fork,table))
   then ins(on(fork,table,D)).
 6. if bel(conversation(Id,i)) then pickUp(fork,D) + send(Id:X, .hold(fork)).
 7. if bel(conversation(Id,i), hold(fork,left), hold(fork,right), neighbours(X,Y))
      bel(X,not(on(fork,table))), bel(Y,not(on(fork,table)))
   then close(Id).
 8. if bel(conversation(Id,X)), goal(X, hold(fork))
   then putDown(fork,D) + send(Id:X, .on(fork,table), not(hold(fork))).
 9. if bel(conversation(Id,X), neighbour(X,D)), bel(X, hold(fork))
   then del(on(fork,table,D)) + send(Id:X, ?on(fork,table)).
}
action-spec{
   think{pre{not(hungry)}post{hungry}}
   pickUp(fork,D){pre{on(fork,table,D)}post{hold(fork,D),not(on(fork,table,D))}}
   eat{pre{hungry,hold(fork,left), hold(fork,right)}post{not(hungry)}}
   putDown(fork, D){pre{hold(fork,D)}post{on(fork,table,D),not(hold(fork,D))}}
}
}
```

6 Conclusion

In this paper, we have introduced an alternative semantics for communication in agent programming languages, based on the idea that a received message can be used to (re)construct a mental model of the sender. We have made this idea precise for the GOAL agent programming language. Also, we have introduced the concept of a conversation to synchronize actions and communication in a multi-agent system. We have shown how these new constructs can be used to program a solution for a classic problem in concurrency theory. We are currently implementing these ideas to allow further experimentation and testing.

References

1. Austin, J.: How to Do Things with Words. Oxford University Press, London (1962)
2. Searle, J.: Speech acts. Cambridge University Press (1969)
3. Singh, M.: A social semantics for agent communication languages. In: Issues in Agent Communication, Springer-Verlag (2000) 31–45
4. Chopra, A., Singh, M.: Constitutive interoperability. In: Proceedings of the 7th International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS'08). (2008) 797–804

5. Labrou, Y., Finin, T.: A semantics approach for KQML - a general purpose communication language for software agents. In: Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94), ACM (1994)
6. FIPA: Fipa communicative act library specification. Technical Report SC00037J, Foundation for Intelligent Physical Agents, Geneva, Switzerland (2002)
7. Singh, M.: Agent Communication Languages: Rethinking the Principles. IEEE Computer **31**(12) (1998) 40–47
8. Vieira, R., Moreira, A., Wooldridge, M., Bordini, R.: Formal Semantics of Speech-Act Based Communication in an Agent-Oriented Programming Language. Artificial Intelligence Research **29** (2007) 221–267
9. Wooldridge, M.: Semantic Issues in the Verification of Agent Communication Languages. Autonomous Agents and Multi-Agent Systems **3**(1) (2000) 9–31
10. de Boer, F., Hindriks, K., van der Hoek, W., Meyer, J.J.: A Verification Framework for Agent Programming with Declarative Goals. Journal of Applied Logic **5**(2) (2007) 277–302
11. Dastani, M.: 2APL: a practical agent programming language . Journal Autonomous Agents and Multi-Agent Systems **16**(3) (2008) 214–248
12. Bulling, N., Hindriks, K.V.: Communicating Rational Agents: Semantics and Verification. (2009) submitted.
13. Hindriks, K.V., Boer, F.S.D., Hoek, W.V.D., Meyer, J.J.C.: Semantics of communicating agents based on deduction and abduction. In: Issues in Agent Communication, Springer Verlag (2000) 63 – 79
14. Wooldridge, M.: An introduction to multiagent systems. John Wiley and Sons, LTD, West Sussex (2002)
15. Gärdenfors, P.: Knowledge in Flux: Modelling the Dynamics of Epistemic States. MIT Press (1988)
16. Ben-Ari, M.: Principles of Concurrent and Distributed Programming. Prentice Hall (1990)

A Goal program for the dining philosophers

```
main i { % i, a number between 1 and N, is the name of the philosopher agent
  knowledge{
    neighbour(X,left) :- i>1, X is i-1.
    neighbour(X,left) :- i=1, X is N.     % N is the number of philosophers.
    neighbour(X,right) :- i<N, X is i+1.
    neighbour(X,right) :- i=N, X is 1.
    neighbours(X,Y) :- neighbour(X,left), neighbour(Y,right).
    forkAvailable(D) :- hold(fork,D) ; on(fork,table,D).
    forksAvailable :- forkAvailable(left), forkAvailable(right).
  beliefs{ hold(fork,left). }
  goals{ hold(fork,left), hold(fork,right). }

  program{
    if true then think.          % can only think when not hungry (see action spec)
    if true then eat.            % can only eat when hungry and holding forks

    if bel(hungry) then adopt(hold(fork,left), hold(fork,right)).

    % Initiate conversation with neighbors if you want to eat but forks are not
    % available by sending an imperative: See to it that I hold the fork.
    if goal(hold(fork,_)), bel(not(forksAvailable), neighbours(X,Y))
      then send(new:{X,Y},!hold(fork)).

    % Ongoing conversation initiated by philosopher itself.
    % Only in this case the philosopher will pick up forks.
    if bel(neighbour(X,D), not(hold(fork,D))), bel(X, on(fork,table))
      then ins(on(fork,table,D)).
    if bel(conversation(Id,i)) then pickUp(fork,D) + send(Id:X, .hold(fork)).
    % Close the conversation if I hold both forks and neighours have noticed this.
    if bel(conversation(Id,i), hold(fork,left), hold(fork,right), neighbours(X,Y))
        bel(X,not(on(fork,table))), bel(Y,not(on(fork,table)))
      then close(Id).

    % Ongoing conversation initiated by a neighbouring philosopher
    % Only in this case a philosopher will put down a fork.
    if bel(conversation(Id,X)), goal(X, hold(fork))
      then putDown(fork,D) + send(Id:X, .on(fork,table), not(hold(fork))).
    if bel(conversation(Id,X), neighbour(X,D)), bel(X, hold(fork))
      then del(on(fork,table,D)) + send(Id:X, ?on(fork,table)).
  }
  action-spec{
     think{pre{not(hungry)}post{hungry}}
     pickUp(fork,D){pre{on(fork,table,D)}post{hold(fork,D),not(on(fork,table,D))}}
     eat{pre{hungry,hold(fork,left), hold(fork,right)}post{not(hungry)}}
     putDown(fork, D){pre{hold(fork,D)}post{on(fork,table,D),not(hold(fork,D))}}
  }
}
```

Introducing Relevance Awareness in BDI Agents

Emiliano Lorini[1] and Michele Piunti[2]

[1] Institut de Recherche en Informatique de Toulouse (IRIT), France
[2] Università degli studi di Bologna - DEIS, Bologna, Italy

Abstract. Artificial agents engaged in real world applications require accurate allocation strategies in order to better balance the use of their bounded resources. In particular, they should be capable to filter out all irrelevant information and just to consider what is relevant for the current task that they are trying to solve. The aim of this work is to propose a mechanism of relevance-based belief update to be implemented in a BDI cognitive agent. This in order to improve the performance of agents in information-rich environments. In the first part of the paper we present the formal and abstract model of the mechanism. In the second part we present its implementation in the *Jason* platform and we discuss its performance in simulation trials.

1 Introduction

Realistic cognitive agents are by definition resource-bounded [4], hence they should not waste time and energy in reasoning, fixing and reconsidering their knowledge on the basis of every piece of information they get. For this reason, they require accurate allocation strategies in order to better balance the use of their bounded computational resources. In this paper we present a computational model of a mechanism of relevance-based belief update. This mechanism is responsible for filtering out all non-relevant information and for considering only what is *relevant* for the current task that an agent is trying to solve. We show how such a mechanism can be implemented in a BDI (Belief, Desire, Intention) agent [19]. The BDI is a well-established framework which is aimed at describing an agent's mental process of deciding, moment by moment on the basis of current beliefs, which action to perform in order to achieve some goals. The mechanism we propose will accomplish the following general function in an agent reasoning process: (*i*) to signal the inconsistency between the agent's beliefs and an incoming input which is relevant with respect to the agent's current intentions and (*ii*) to trigger a process of belief update in order to integrate such a relevant input in the agent's belief base. More generally, we suppose that at each moment an agent is focused and allocates his attentive resources on a particular task that he is trying to fulfill and on a certain number of intentions which represent the pragmatic solution selected by the agent to accomplish the task [2]. In so doing, the agent ignores all incoming input which is not relevant with respect to the current task on which he is focused and only considers those information which are relevant. On the contrary, if a relevant input turns out to be incompatible with respect to the pre-existent beliefs of the agent, the agent reconsiders them.

The model of a mechanism of relevance-based belief update proposed in this paper is also intended to bridge the existing gap between formal and computational models of

belief change and cognitive models of belief dynamics. Indeed, formal approaches to belief change implicitly assume that when an agent perceives some fact such a perception is always a precursor of a process of belief change. On the contrary, we model here these precursors of a process of belief change and, in agreement with cognitive theories of bounded rationality (e.g. [16]), we show that implementing them in a resource-bounded agent can improve his performance in information-rich environments requiring massive perceptive activities. Moreover, this proposal is intended to provide a novel understanding of the design and implementation of cognitive agents. Whereas mainstream agent platforms provide sophisticated mechanisms to process messages and Agent Communication Languages (ACLs), the perception of heterogeneous events and information is often shaped on the basis of technical constructs which are defined in a domain-dependent fashion and constrained with constructs at the language level. Mainstream programming models are not sufficiently flexible to integrate alternative perceptive capabilities of agents, and to relate them with cognitive concepts such as the ones of the BDI approach. On the contrary, we present in this paper a cognitive model in which the relationship between perception and goal processing is clearly specified in terms of the pivotal notion of relevance.

The paper is organized as follows. Section 2 provides the definition of an agent's abstract model. Section 3 applies the agent's abstract model to the formalization of a specific problem domain. In section 4 the cognitive architectures of two general typologies of BDI agents are formally defined —respectively implementing traditional BDI interpreter and BDI^{rel}with relevance awareness abilities. Section 5 describes a programming model for the BDI^{rel}agent, discussing how it has been implemented with the *Jason* platform [1]. Finally, section 6 compares the performance of agents engaged in simulated experiment in the scenario previously described.

2 An agent's abstract model

In this section a definition of an agent's abstract model is given. This includes data obtained by perceptions, static causal knowledge, volatile beliefs, desires and intentions, desire-generating and planning rules, and a repertoire of basic actions.

Let **VAR**$=\{X_1,\dots,X_n\}$ be a non-empty set of random variables. We suppose that each random variable $X_i \in$ **VAR** takes values from a non-empty set of variable assignments Val_{X_i}. For each set Val_{X_i} we denote by $Inst_{X_i}$ the corresponding set of all possible instantiations of random variable X_i. For example, suppose that $Val_{X_i}=\{x_1,\dots,x_r\}$ then $Inst_{X_i}=\{X_i{=}x_1,\dots,X_i{=}x_r\}$. We denote by $Inst$ the set of all possible instantiations of all random variables, that is: $Inst=\bigcup_{X_i\in\mathbf{VAR}} Inst_{X_i}$.

Perceived data $\Gamma \subseteq Inst$ is a set of perceived data which fixes the value of certain variables that an agent perceives at a certain moment. For example, $\Gamma=\{X_i{=}x\}$ means "the agent perceives the event $X_i{=}x$". We denote by

$$\Gamma_{Var}=\{X_i \in \mathbf{VAR} \mid \exists x \in Val_{X_i} \text{ such that } X_i{=}x \in \Gamma\}$$

the subset of **VAR** which includes the variables that an agent observes at a certain moment. Here we suppose that for all $X_i \in \Gamma_{Var}$, $Inst_{X_i} \cap \Gamma$ is a singleton, that is, we suppose that an agent cannot perceive two different instantiations of the same vari-

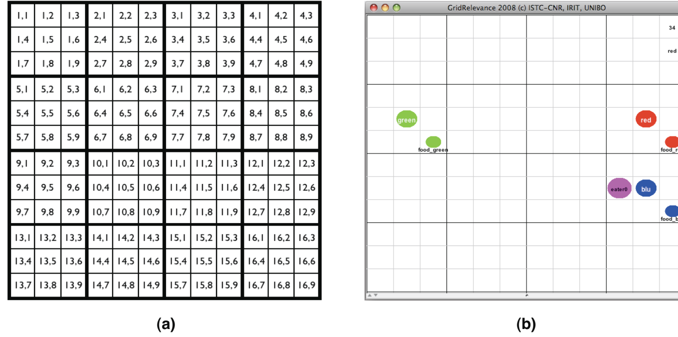


Fig. 1. Environment grid in agent's domain representation (a) and the running experiment (b).

able. We use the notation $\Gamma(X_i)$ to denote this singleton for any $X_i \in \Gamma_{Var}$, that is, $\Gamma(X_i) = Inst_{X_i} \cap \Gamma$.

Stable causal knowledge K is a Bayesian network which represents the joint probability distribution over the set of random variables **VAR**. A Bayesian network is a directed acyclic graph (DAG) whose nodes are labeled by the random variables in **VAR** and the edges represent the causal influence between the random variables in **VAR** [14]. Given an arbitrary random variable X (i.e. an arbitrary node) in the Bayesian network K we denote by $anc(X)$ the set of ancestors of X. Formally, Z is an *ancestor* of X in the Bayesian network K if there is a directed path from Z to X in K.

Moreover, given an arbitrary random variable X in the Bayesian network K, we denote by $par(X)$ the set of parents of X in the Bayesian network. Formally, Z is a *parent* of X in the Bayesian network K if Z is an ancestor of X in K which is directly connected to Z. Finally, we associate to each random variable X in K a conditional probability distribution $P(X \mid par(X))$. The Bayesian network K encodes the agent's causal knowledge of the environment. Here we suppose that this part of the agent's knowledge is stable and can not be reconsidered.

Volatile beliefs An agent's abstract model also includes beliefs that can change over time, i.e. the agent's volatile beliefs [3]. Given a random variable $X_i \in$ **VAR**, we denote by $\sum_{X_i}$ the set of all possible probability distributions over the random variable X_i. We denote by BEL the cartesian product of all $\sum_{X_i}$, that is $\mathsf{BEL} = \prod_{X_i \in \mathbf{VAR}} \sum_{X_i}$. BEL includes all possible combinations of probability distributions over the random variables in **VAR**. Elements in BEL are vectors $\mathsf{B} = \langle \mathsf{B}_1, \ldots, \mathsf{B}_n \rangle, \mathsf{B}' = \langle \mathsf{B}'_1, \ldots, \mathsf{B}'_n \rangle, \ldots$. Every vector B corresponds to a particular configuration of beliefs of the agent. In this sense, BEL includes all potential configurations of beliefs of the agent.

Suppose that $Val_{X_i} = \{x_1, \ldots, x_r\}$. Then, every element B_i in a configuration of beliefs B is just a set $\{(X_i{=}x_1){=}a_1, \ldots, (X_i{=}x_r){=}a_r\}$ of probability assignments $a_1, \ldots, a_r \in [0, 1]$ to each possible instantiations of the variable X_i.

Given a specific configuration of beliefs $\mathsf{B} = \langle \mathsf{B}_1, \ldots, \mathsf{B}_n \rangle$, we write $\mathsf{B}(X_i{=}x){=}a$ if and only if $(X_i{=}x){=}a \in \mathsf{B}_i$. For example, $\mathsf{B}(X_i{=}x){=}0.4$ means that given the configuration of beliefs $\mathsf{B} = \langle \mathsf{B}_1, \ldots, \mathsf{B}_n \rangle$ the agent assigns probability 0.4 to the fact that variable X_i takes value x. Moreover, we denote by $\mathsf{B}(X_i{=}x)$ the number $a \in [0, 1]$ such that $\mathsf{B}(X_i{=}x){=}a$.

Intentions and desires We also model motivational attitudes by denoting with INT the set of potential intentions of an agent. Here we suppose that every instantiation of a variable in $Inst$ is a potential intention of the agent. We denote by $\mathsf{I}, \mathsf{I}', \ldots \in 2^{\mathsf{INT}}$ specific sets of intentions of the agent. Given a specific set of intentions of the agent I, we denote by I_{Var} the subset of **VAR** which includes all intended variables, that is, all those variables which have (at least) one instantiation in I. Formally:

$$\mathsf{I}_{Var} = \{X_i \in \mathbf{VAR} \mid \exists x \in Val_{X_i} \text{ such that } X_i = x \in \mathsf{I}\}.$$

We call *intention supports* all variables that are parents in the Bayesian network K of some intended variable. The set of intention supports is formally defined as follows:

$$\mathsf{SUPP}_{Var} = \{X_i \in \mathbf{VAR} \mid \exists X_j \in \mathsf{I}_{Var} \text{ such that } X_i \in par(X_j)\}.$$

Note that the set of intention supports includes intention preconditions, that is, all conditions on which the achievement of an intended result of the agent depends. DES is the set of all potential desires of the agent. As for intentions, we suppose that every instantiation of a variable in $Inst$ is a potential desire of the agent, that is, $\mathsf{DES}=Inst$. We denote by $\mathsf{D}, \mathsf{D}', \ldots \in 2^{\mathsf{DES}}$ specific sets of desires of the agent.

Desire-generating rules and planning rules We specify a set DG of desire-generating rules and a set PL of planning rules. A desire-generating rule in DG is a desire-generation rule in the style of [7] of the form:

$$\psi_1, \ldots, \psi_s \mid \lambda_1, \ldots, \lambda_j \Longrightarrow \varphi_1, \ldots, \varphi_t.$$

Such a rule is responsible for generating t desires $\varphi_1, \ldots, \varphi_t$ when the agent has s beliefs $\psi_1, \ldots, \psi_s$ and j intentions $\lambda_1, \ldots, \lambda_j$. The set of desire-generating rules DG corresponds to a function $options : \mathsf{BEL} \times 2^{\mathsf{INT}} \mapsto 2^{\mathsf{DES}}$. This function returns a specific set D of desires, given a specific configuration B of beliefs and a specific set I of intentions.

A planning rule in the set of planning rules PL is a plan-generating rule of the form:

$$\psi_1, \ldots, \psi_s \mid \lambda_1, \ldots, \lambda_j \Longrightarrow \alpha_1, \ldots, \alpha_t.$$

Such a rule is responsible for generating t plans $\alpha_1, \ldots, \alpha_t \in \mathsf{ACT}$, where ACT is the repertoire of actions and plans of our agent, when the agent has s beliefs $\psi_1, \ldots, \psi_s$ and j intentions $\lambda_1, \ldots, \lambda_j$. The set of planning rules PL corresponds to a function $plan : \mathsf{BEL} \times 2^{\mathsf{INT}} \mapsto 2^{\mathsf{ACT}}$. This function returns a set π of plans, given a specific set B of beliefs and specific set I of intentions. To summarize, an agent's abstract model is defined as a tuple $\langle \Gamma, \mathsf{K}, \mathsf{B}, \mathsf{D}, \mathsf{I}, \mathsf{DG}, \mathsf{PL}, \mathsf{ACT} \rangle$, where each element in the tuple is defined as before.

3 Formalization of the experimental scenario

Our experimental scenario is represented by the 12×12 grid in Fig. 1(a). An agent moves in the grid being driven by the goal of finding fruits of a certain color, according to the ongoing season. Indeed, agents look for fruits of different colors in different seasons of the year. We suppose that there are three different seasons and related colors of fruits and trees: the red season, the blue season and the green season. Agents are intrinsically motivated to look for and to eat red fruits during the red season, blue fruits during the blue season and green fruits during the green season. Environmental dynamics are characterized by periodic season cycles: after s_t rounds the season changes on

BDI^{rel} **agent control loop**	BDI **agent control loop**
1. B:=B_0;	1. B:=B_0;
2. I:=I_0;	2. I:=I_0;
3. while (true) do	3. while (true) do
4. get new percept Γ;	4. get new percept Γ;
5. if $\mathbf{R}(\mathsf{I},\Gamma,\mathsf{B}) > \Delta$ then	5. B:=$bu(\Gamma,\mathsf{B})$;
6. B:=$bu^{*}(\Gamma,\mathsf{B},\mathsf{I})$;	6. D:=$options(\mathsf{B},\mathsf{I})$;
7. end-if	7. I:=$filter(\mathsf{B},\mathsf{D},\mathsf{I})$;
8. D:=$options(\mathsf{B},\mathsf{I})$;	8. π:=$plan(\mathsf{B},\mathsf{I})$;
9. I:=$filter(\mathsf{B},\mathsf{D},\mathsf{I})$;	9. $execute(\pi)$;
10. π:=$plan(\mathsf{B},\mathsf{I})$;	10. end-while
11. $execute(\pi)$;	
12. end-while	

Table 1. Abstract interpreter implemented by the two typologies of agents

the basis of a periodic function and the intrinsic motivation of an agent changes accordingly. Fruits of any color occupy cells (i, j) (with $1 \leq i \leq 16$ and $1 \leq j \leq 9$), where i indicates the number of the macro-area in the grid and j the number of the cell inside the macro-area. Trees of any color occupy macro-areas i of size 3×3 (with $1 \leq i \leq 16$) in the grid depicted in Fig. 1(a). We suppose that at each moment for every color there is exactly one fruit and tree of that color in the grid. Moreover, we suppose an objective dependence between trees and fruits in the grid: a fruit of a certain color is a sign of the presence of a fruit of the same color in the immediate neighborhood. Agents exploit these signs during their search of fruits. We suppose that a tree of any color is randomly placed in a macro-area i of size 3×3. Given a tree of a certain color in a macro-area i of size 3×3, a fruit of the same color is randomly placed by the simulator in one of the four cells inside the macro-area i. For example, if a red tree is in the macro-area 1 of the grid then for each cell $(1, i)$ with $1 \leq i \leq 9$, there is $\frac{1}{9}$ of probability that a red fruit is located in that cell. Fruits and trees change periodically their positions. The dynamism factor δ indicates how many seasons have to pass before a tree location changes.

We impose constraints on the perceptual capabilities of agents by supposing that an agent sees only those fruits which are in the cells belonging to the same macro-area in which the agent is. For example, if the agent is in cell $(6, 1)$, he only sees those fruits which are in the cells belonging to the macro-area 6. Moreover we suppose that an agent sees only those trees which are situated in the same macro-area in which the agent is or in the four neighboring macro-areas on the left, right up or down. For example, if the agent is in cell $(6, 1)$, he only sees those trees which are in macro-areas $2, 5, 7$ or 10.

The knowledge of our agents is encoded by means of the following eight random variables **VAR**$= \{SEAS, POS, RF, BF, GF, RT, BT, GT\}$. The variables RF, BF, GF, POS take values from the sets $\{(i, j) \mid 1 \leq i \leq 16, 1 \leq j \leq 9\}$, whilst the variables RT, BT, GT take values from the set $\{i \mid 1 \leq i \leq 16\}$. Finally, $SEAS$ takes value from the set $\{red, blue, green\}$. Variables RF, BF, GF specify respectively the position of a red/ blue/ green fruit in the grid depicted in Fig. 1 (a). Variables RT, BT, GT specify respectively the position of a red/blue/green tree in the grid. For example, RT=13 means “there is a red tree in the macro-area 13”. Variable $SEAS$ specifies the current season. For example, $SEAS$=$blue$ means “it is time to look for blue fruits!”. Finally, Variable POS specifies the position of the agent in the grid.

The variables in **VAR** are organized in the Bayesian network K as follows: $par(POS){=}\{\emptyset\}, par(SEAS){=}\{\emptyset\}, par(RT){=}\{\emptyset\}, par(BT){=}\{\emptyset\}, par(GT){=}\{\emptyset\}$, $par(RF){=}\{RT\}, par(BF){=}\{BT\}, par(GF){=}\{GT\}$. Since there are 144 possible positions of a fruit and 16 possible positions of a tree in the grid, each conditional probability table associated with $P(RF \mid RT)$, $P(BF \mid BT)$ and $P(GF \mid GT)$ has $144 \times 16{=}2304$ entries. We suppose that the knowledge an agent has about the dependencies between trees and fruits perfectly maps the objective dependencies between trees and fruits. Hence, we only specify for each tree of a certain color – and arbitrary macro-area $i \in \{1, \ldots, 16\}$ in the grid in which a tree can appear – the 9 conditional probabilities that a fruit of the same color appears in one cell in that macro-area. We suppose for each of them the same probability value $\frac{1}{9}$. All other conditional probabilities have value 0, that is, given a tree of a certain color which appears in an arbitrary macro-area $i \in \{1, \ldots, 16\}$, the probability that there is a fruit of the same color outside that macro-area is zero. More precisely, we have that for all $1 \leq i, j \leq 16$ and $1 \leq z \leq 9$: *(i)* if $i{=}j$ then $P(RF{=}(j, z) \mid RT{=}i){=}\frac{1}{9}$; *(ii)* if $i \neq j$ then $P(RF{=}(j, z) \mid RT{=}i){=}0$. Desire-generating rules in DG are exploited by agents for solving the general task of finding a fruit of a certain color in the grid. Agents are endowed with three general classes of desire-generating rules. The first class includes desire-generating rules of the following form. For $i \in Val_{SEAS}$: $(SEAS{=}i){=}1 \Longrightarrow SEAS{=}i$. These desire-generating rules are responsible for changing the intrinsic motivation of an agent, according to the season change, that is: if an agent is certain that it is time to look for fruits of kind i, then he should form the desire to look for fruits of kind i.

The second class includes desire-generating rules shown in Tab. 2 (DG *Group 1*). These are responsible for orienting the search toward a certain macro-area, according to the current season (i.e. an intention to find fruits of a certain color) and his beliefs about the position of trees in the grid. For instance, if an agent is certain that there is a red tree in the macro-area 3 of the grid (i.e. $(RT{=}3){=}1$) and desires to find a red fruit (i.e. $SEAS{=}red$), then he should form the intention to reach that position of a red tree (i.e. $RT{=}3$). Finally, agents are endowed with desire-generating rules shown in Tab. 2 (DG *Group 2*). These desire-generating rules are responsible for orienting the search of an agent toward a certain cell, according to the current season (i.e. an intention to find fruits of a certain color) and his beliefs about the position of fruits in the grid. For example, if an agent desires to find a blue fruit (i.e. $SEAS{=}blue$) and knows/ is certain that there is a blue fruit in cell $(10, 1)$ of the grid (i.e. $(BF{=}(10, 1)){=}1$), then he should form the intention to move toward that position of the blue fruit (i.e. $BF{=}(10, 1)$).

We suppose that agents have four basic actions $MoveDown$, $MoveUp$, $MoveLeft$ and $MoveRight$ in repertoire. Indeed, at each round they can only move from one cell to the next one. Planning rules encode approaching policies which depend on the agent's current intentions and his actual position in the grid. Agents have both planning rules for reaching macro-areas in the grid (given their current positions) and planning rules for reaching cells in the grid (given their current positions). The latter planning rules are exploited for the local search of a fruit of a certain color inside a macro-area. An example of these planning rule is the following: $(POS{=}(15, 1)){=}1 \mid RT{=}3 \Longrightarrow MoveUp$. Thus, if an agent intends to reach position 3 of a red tree and is certain to be in cell $(15, 1)$ then he should form the plan to move one step up.

DG **Group 1**	DG **Group 2**
For $1 \leq i \leq 16$:	For $1 \leq i \leq 16$ and $1 \leq j \leq 9$:
$(RT{=}i){=}1 \mid SEAS{=}red \Longrightarrow RT{=}i$	$(RF{=}(i,j)){=}1 \mid SEAS{=}red \Longrightarrow RF{=}(i,j)$
$(BT{=}i){=}1 \mid SEAS{=}blue \Longrightarrow BT{=}i$	$(BF{=}(i,j)){=}1 \mid SEAS{=}blue \Longrightarrow BF{=}(i,j)$
$(RT{=}i){=}1 \mid SEAS{=}green \Longrightarrow GT{=}i$	$(RF{=}(i,j)){=}1 \mid SEAS{=}green \Longrightarrow GF{=}(i,j)$

Table 2. Means End rules governing intention selection

4 Relevance and Belief Update

In this section we present two different architectures and corresponding typologies of cognitive agents. The first type of agent corresponds to a standard BDI agent whose control loop is described in the right column of Tab. 1. The second type of agent, whose control loop is described in the left column of Tab. 1, is a BDI agent endowed with a relevance-based mechanism of belief update. We call this second type of agent BDI^{rel}agent.

The formal description of the control loop of the standard BDI agent is similar to [19]. In lines 1-2 the beliefs and intentions of the agent are initialized. The main control loop is in lines 3-10. In lines 4-5 the agent perceives some new facts Γ and updates his beliefs according to a function bu. In line 6 the agent generates new desires by exploiting his desire-generating rules. In line 7 he deliberates over the new generated desires and his current intentions according to the function $filter$.[3] Finally, in lines 8-9 the agent generates a plan for achieving his intentions by exploiting his planning rules and he executes an action of the current plan. The main difference between the standard BDI agent and the BDI^{rel}agent is the belief update part in the control loop. We suppose that a process of belief update is triggered in the BDI^{rel}agent only if the agent perceives a fact and evaluates this as relevant with respect to what he intends to achieve (line 5 in the control loop of the BDI^{rel}agent). In this sense, the BDI^{rel}is endowed with a cognitive mechanism of relevance-based belief update. In fact, this mechanism filters out all perceived facts that are irrelevant with respect to the current intentions. Thus, the BDI^{rel}agent only updates his beliefs by inputs which are relevant with respect to his current intentions. Differently, at each round the standard BDI agent updates his beliefs indiscriminately: for any fact he perceives, he updates his beliefs whether the perceived fact is relevant with respect to his intentions or not.

In order to design the mechanism of relevance-based belief update, we define a notion of local relevance of an input Γ with respect to an intention $Y{=}y \in \mathsf{I}$, given the configuration of beliefs B. This is denoted by $r(Y{=}y,\Gamma,\mathsf{B})$ and is defined as follows.

$$r(Y{=}y,\Gamma,\mathsf{B}) = \begin{cases} \Rightarrow \text{ If } Y \in \Gamma_{Var}: \\ 1 - \mathsf{B}(\Gamma(Y)) \\ \\ \Rightarrow \text{ If } par(Y) \subseteq \Gamma_{Var} \text{ and } Y \notin \Gamma_{Var}: \\ \|\mathsf{B}(Y{=}y) - P(Y{=}y \mid \{X_i{=}x \mid \\ X_i \in par(Y) \text{ and } X_i{=}x \in \Gamma\})\| \\ \\ \Rightarrow \text{ If } par(Y) \not\subseteq \Gamma_{Var} \text{ and } Y \notin \Gamma_{Var}: \\ 0 \end{cases} \quad (1)$$

[3] Space restrictions prevent a formal description of the function $filter$ here (see [19] for a detailed analysis). Only notice that this function is responsible for updating the agent's intentions with his previous intentions and current beliefs and desires (i.e. $filter : \mathsf{B} \times 2^{\mathsf{I}} \times 2^{\mathsf{D}} \mapsto 2^{\mathsf{I}}$).

The degree of local relevance of the percept Γ with respect to intended fact $Y{=}y \in \mathsf{I}$ (given the agent's configuration of beliefs B) is defined on the basis of three conditions.

According to the first condition, if the intended variable Y is also a perceived variable in Γ_{Var} (i.e. there exists an instantiation of Y which is an element of Γ) then, $r(Y{=}y,\Gamma,\mathsf{B})$ is equal to the degree of unexpectedness of the percept Γ (i.e. $1-\mathsf{B}(\Gamma(Y))$). The degree of unexpectedness of the percept Γ is inversely proportional to the prior probability assigned by the agent to the perceived instantiation of the intended variable Y (see [11] for an analysis of the notion of unexpectedness).

According to the second condition, if the intended fact $Y{=}y$ is not an instantiation of a perceived variable in Γ_{Var} and the parents of Y in the Bayesian network K are perceived variables in Γ_{Var} then, $r(Y{=}y,\Gamma,\mathsf{B})$ is equal to the degree of discrepancy between the intended fact $Y{=}y$ and the percept Γ. The degree of discrepancy between the intended fact $Y{=}y$ and the percept Γ is given by the absolute value of the difference between the probability assigned to $Y{=}y$ (i.e. $\mathsf{B}(Y{=}y)$) and the conditional probability that $Y{=}y$ is true given that the perceived instantiations of the parents of Y are true (i.e. $P(Y{=}y \mid \{X_i{=}x \mid X_i \in par(Y)$ and $X_i{=}x \in \Gamma\})$).

According to the third condition, if the intended fact $Y{=}y$ is not an instantiation of a perceived variable in Γ_{Var} and there is some parent of Y in the Bayesian network K that is not a perceived variable in Γ_{Var} then $r(Y{=}y,\Gamma,\mathsf{B})$ is zero. This third condition corresponds to the irrelevance of the incoming input Γ with respect to the agent's intention $Y{=}y$. Under this third condition, the agent simply ignores the input.

Let us now define a notion of global relevance, noted $\mathbf{R}(\mathsf{I}, \Gamma, \mathsf{B})$, as the maximum value of local relevance for each intended fact $Y{=}y \in \mathsf{I}$:

$$\mathbf{R}(\mathsf{I}, \Gamma, \mathsf{B}) = \max_{Y=y \in \mathsf{I}} r(Y{=}y,\Gamma,\mathsf{B}) \qquad (2)$$

This notion of global relevance is used in the control loop of the BDI^{rel}agent: if the new percept Γ is responsible for generating a degree of global relevance higher than Δ (with $\Delta \in [0, 1]$) then a process of belief update is triggered and the BDI^{rel}agent adjusts his beliefs with the perceived data Γ according to a function bu^*. The belief update function bu^* of the BDI^{rel}agent takes in input the set of intentions I, the belief configuration B and the percept Γ and returns an update belief configuration B', that is: $bu^* : 2^{Inst} \times \mathsf{BEL} \times 2^{\mathsf{INT}} \mapsto \mathsf{BEL}$.

More precisely, suppose that $bu^*(\Gamma, \mathsf{B}, \mathsf{I}){=}\mathsf{B}'$. The set B' is defined according to the following three conditions. For every $Y \in \mathbf{VAR}$ we have:

1. $\Rightarrow$ If $Y \in \mathsf{I}_{Var}$ or $Y \in \mathsf{SUPP}_{Var}$, and $Y \in \Gamma_{Var}$:
   $\mathsf{B}'(\Gamma(Y)){=}1$ and for every $Y{=}x \in Inst_Y \setminus \Gamma(Y)$, $\mathsf{B}'(Y{=}x){=}0$.
2. $\Rightarrow$ If $Y \in \mathsf{I}_{Var}$ or $Y \in \mathsf{SUPP}_{Var}$, and $par(Y) \subseteq \Gamma_{Var}$ and $Y \notin \Gamma_{Var}$:
   for every $Y{=}y \in Inst_Y$, $B'(Y{=}y){=}P(Y{=}y \mid \{X_i{=}x \mid X_i \in par(Y)$ and $X_i{=}x \in \Gamma\})$.
3. $\Rightarrow$ Otherwise:
   for every $Y{=}y \in Inst_Y$, $B'(Y{=}y){=}B(Y{=}y)$.

According to the previous formal characterization of the function bu^*, the BDI^{rel}agent only reconsiders the probability distributions over his intentions $Y \in \mathsf{I}_{Var}$ and over his intention supports $Y \in \mathsf{SUPP}_{Var}$. In fact, we suppose that the BDI^{rel}agent only reconsiders those beliefs which are directly related with his intentions or his intention supports, since he allocates his attention on the current task he is trying to solve. More precisely: if Y is either an intended random variable in I_{Var} or an intention support

in SUPP_{Var}, and Y is a perceived variable in Γ_{Var}, then the updated probability distribution over Y assigns probability 1 to the perceived instantiation $\Gamma(Y)$ of variable Y and probability 0 to all the other instantiations of variable Y (condition 1); if Y is either an intended random variable in I_{Var} or an intention support in SUPP_{Var}, Y is not a perceived variable in Γ_{Var}, but Y's parents in the Bayesian network are perceived variables in Γ_{Var}, then the updated probability distribution over Y assigns to each instantiations $Y{=}y$ of variable Y a probability which is equal to the conditional probability that $Y{=}y$ is true given that the perceived instantiations of the parents of Y are true (i.e. $P(Y{=}y \mid \{X_i{=}x \mid X_i \in par(Y)$ and $X_i{=}x \in \Gamma\})$ (condition 2). In all other cases the probability distribution over Y is not updated.

Space restrictions prevent a formal description of the belief update function bu of the standard BDI agent. Let us only say that function bu (differently from the function bu^* of the BDI^{rel} agent) updates indiscriminately all beliefs of the agent, that is, at each round the standard BDI agent reconsiders the probability distributions over all random variables $Y \in \mathbf{VAR}$.[4]

5 Programming Model

This section introduces the programming model implementing the mechanism of relevance-based belief update described above. The experiment platform has been built on top of CARTAGO, a framework for developing MAS environments based on the abstraction of agents and artifacts [13]. Agents architectures have been implemented using *Jason*, a programming platform for BDI agents based on AgentSpeak(L) [1].

Environment The rationale behind the adoption of the Agents and Artifacts (A&A) meta-model for the design of the experimental platform resides in the particular interaction model provided by CARTAGO, where all the mechanisms related to agent's perception and action, are regulated by the framework at a system level. Agents – independently from their internal model and technology– can play in CARTAGO environments by interacting with artifacts through operation of *use* which consists in exploiting the artifact's usage interface. Besides, agent's perceptive activities are defined through the notions of *observation* which consists in retrieving the information that artifacts display, and *perception*, enabling agents to sense signals and events coming from artifacts. In this perspective artifacts can be conceived as a target for agents' overall activity and thus exploited to provide information to agents in a machine-readable format. Once an artifact observable property or a signal is perceived by an agent, it may become a perceived datum (i.e. a percept). Following the basic idea provided in this work, only if such a percept is relevant, it can used to update agent's belief base. To implement the scenario described in section 3, the environment has been instrumented with three different artifacts.

- `Timer` provides agents with timing information and enables the automatic mechanisms regulating the dynamism of the environment. Accordingly, it makes available

[4] Function bu has the same conditions of function bu^* specified above. The only difference is that in bu the requirement '$Y \in \mathsf{I}_{Var}$ or $Y \in \mathsf{SUPP}_{Var}$' is not specified.

two properties which are related to its internal state: `ticktime` (indicating the actual value of simulated time) and `season` (indicating the value of the ongoing season).

– `GridBoard` provides pragmatic operations to be used as actions by agents (i.e. $Move$, Eat) and feedback information about the effects of these actions in the environment. In addition, based on the temporal signals generated by the Timer and on the actions performed by the agents, it provides the logic governing the topology and the physical consistency of the overall environment.
– `GridboardGUI` based on the internal states of the previously described artifacts, it provides the graphical user interface for the system.

Agents The overall goal for agents is to find and to eat fruit items. At any time step (i.e., round) the score associated to a fruit depends on the ongoing season: a fruit of a given color (e.g. blue) provides a reward of +1 only if the ongoing season has the same color (e.g. the blue season). It is supposed that a tree changes its position at regular time intervals due to the ongoing dynamism δ, hence agents need to govern their behavior only with respect to the actual situation. In particular, an agent should maintain an updated knowledge of the overall environment in order to avoid wasting resources when looking for areas which are not profitable. In these conditions, an effective strategy is to look for fruits by using trees as reference points. Once a tree which is related to the ongoing season is encountered, the agent can perform epistemic actions aimed at updating his beliefs about the presence of fruits in the macro-area in which the agent is located. At an implementation level, a Bayesian belief base to include the special belief set governing goal deliberation and intention selection has been realized. In particular, the `jason.architecture.AgArch` and `jason.asSemantics.Agent`, which are the kernel units used by the Jason reasoning-engine to percept and update beliefs, have been extended introducing specialized data-types and methods allowing the dynamic query on the probability distribution of domain entities. Thus, an agent uses a working memory realized with a series of dynamic hash-maps. Each hash-map is related to a given season type and can be accessed by indicating the coordinates of a given cell. Once a tree is perceived, the agent can control if any belief relating to that location is present and possibly update it. When an agent updates his belief about the position of a certain tree, he will also update the belief about the position of the fruit of the same color.

The adopted artifact-based environment promotes a principled design for agent perceptive activities. Through the integration of an additional set of internal actions enabling A&A interactions, agents are capable to perceive –and possibly filter– the signals coming from the scrutinized artifacts. Besides, events coming from artifacts can signal to the agent situations which require special attention. These signals are automatically sent back to the agent control system in order to be filtered and processed. Indeed, once some particular signal is encountered, agents can exploit it for updating their beliefs, or for reconsidering their intentions. Therefore, after becoming aware of some relevant facts, agents can elicit belief update or an adaptive re-allocation of their computational resources. Artifacts provide, in this case, two kinds of signals: signals for temporal synchronization (agents rule their actions based on a clock signals perceived from the `Timer`) and signals belonging to the set Γ_{Var}, which in turn contains the percepts corresponding to visible entities. As shown in the following *Jason* cutout, once a clock signal is

received from the focused `Timer`, an internal action `gridworld.perceptRel` interacts with the `GridBoard` to retrieve the list of percepts indicating all the visible entities.

```
+tick_time(X)[source("percept"), artifact("timer"), workspace("Relevance")]
  : actual_season(S)
  <- -+time(X);
     gridworld.perceptRel(S);
     !deliberateTarget.
```

The `gridworld.perceptRel` perceptive action has two different implementations respectively for the BDI agent and for the BDI^{rel}agent. `gridworld.perceptRel` is supposed to realize the belief update functions bu and bu^{*}. Percepts are inserted in the agent working memory and then filtered by the belief update function. In the case of the BDI^{rel}agent, once these percepts are related to the current intention (i.e. actual season) they are stored in the agent memory as permanent belief facts. In particular, the bu^{*} is supposed to retrieve from the `GridBoard` the list of visible entities (these elements become percepts). Moreover, for each retrieved fact, bu^{*} deletes the beliefs actually referring to entities which are not already present in the actual range of sight (trees and fruits can disappear due to the environment dynamism) and adds a new fact to the belief base only if the scrutinized percept in Γ matches the current intention. Notice that in the case of the described scenario (where the agent's intentions depend on the current season) the threshold Δ is set to 0. For discriminating relevant and not relevant percepts a simple pattern matching is used. Hence, the function of local relevance $r(Y{=}y,\Gamma,\mathsf{B})$ is greater than zero when the current season matches the entity type, otherwise $r = 0$.

After the execution of the `perceptRel`, the BDI agent has a complete knowledge of the actual state of its surroundings (i.e. the belief base is supposed to be consistent with the actual state of the visible area). On the other side, BDI^{rel}only considers those information which are relevant with respect to the current situation. Then, to achieve their goals, both agents can adopt the following plans to decide the next course of action.

```
+!deliberateTarget
  : actual_season(S) & food(S,X,Y)
  <- -+targetLoc(X,Y);
     !doAction.
```

```
+!deliberateTarget
   : actual_season(S) & not food(S,_,_)
   & tree(S,X,Y)
   <- -+targetLoc(X,Y);
      !doAction.
+!deliberateTarget <- !doAction.
```

It is worth nothing that, according to the *Jason* transition system, the desire-generating rules described in Tab. 2 are here expressed by means of *context-conditions*, i.e. in form of belief formulae. The belief `targetLoc(X,Y)` can refer to a given fruit location only if the agent has already located a fruit and has stored a related fact in his belief base. If in the belief base there are no facts concerning fruits or trees related to the ongoing season, the agent will perform an epistemic action by exploring the grid in order to discover some new relevant fact. Besides, once beliefs are canceled from the belief base by the internal belief update activities –ruled respectively by bu for the standard BDI agent and bu^{*} for the BDI^{rel}agent– the agent reconsiders his intentions and selects a new target to be reached.

```
-food(T,X,Y) : targetLoc(X,Y)
       <-  !deliberateTarget.
```

```
-tree(T,X,Y) : targetLoc(X,Y)
       <-  !deliberateTarget.
```

In so doing, the BDI agent and the BDI^{rel}agent update their belief base in two circumstances: (*i*) when the actual belief base is wrong (not consistent with the perceived state

	$\delta = 3$		$\delta = 2$		$\delta = 1$	
	BDI	BDI^{rel}	BDI	BDI^{rel}	BDI	BDI^{rel}
$Goal.eff$	42.375	42.375	38.185	37.625	33.937	33.875
$Cost.eff$	92.125	78.437	101.437	85.312	143.125	107.875
$cost.ratio$	2.381	2.012	2.882	2.252	4.379	3.214

Table 3. Amount of achieved goals and performed belief updates measured at the end of the experiment series for BDI and BDI^{rel}agents in environment with dynamism $\delta \in \{1, 2, 3\}$.

of the environment), and (*ii*) when the actual belief base is incomplete (due to a lack of knowledge). Finally, by using the operations allowed by the `GridBoard` interface, and taking into account the planning rules discussed in section 3, a `!doAction` realizes the basic pragmatic actions (i.e., *eat* a fruit or *move* towards `targetLoc(X,Y)`).

6 Experiment

This section discusses a series of experiments comparing the performances for BDI vs. BDI^{rel} agents.

Experiment setting In our experimental setting we suppose for simplicity that agents have always access to their current position in the grid, and that they are always notified of season changes. Therefore, at the beginning of a new season, an agent knows that it is time to look for fruits of a different color, and he adopts the goal to look for fruits of a different color.

Agents' performances have been evaluated according to metrics measuring two basic kinds of effectiveness. Goal effectiveness ($Goal.eff$) represents the total amount of achieved goals during a trial (i.e., eaten fruits), while cost effectiveness ($Cost.eff$) is the total amount of update operations performed by the agent on his belief base. Then we define the $cost.ratio$ of an agent in terms of the agent's belief update cost divided by the total amount of achieved goals ($Cost.eff/Goal.eff$). This gives a quantitative measure of how many units of cost the agent needs to spend for each achieved goal. In other terms, a $cost.ratio = 1$ means that the agents performs a belief update operation for every achieved goal. Besides, since only one $Move$ action is allowed for each time step, the adopted metrics provide insights on how many pragmatic actions are needed for agents to achieve their goals.

The length of experiments has been set to 900 rounds in order to be long enough for the metrics to become stable. So far, the global performance of each agent is measured by averaging $cost.ratio$ of 16 trials in environments with different dynamism δ. Season length s_t is set at 15 rounds, while we consider n=3 seasons (respectively red, blue and green) with three associated types of tree and fruit. The initial placements of entities are randomly selected, while a fruit of a given color is generated at the beginning of each corresponding season. Finally, we assume that no more than one fruit for any given color is present in the grid at the same time.

Discussion Fig. 1 (b) shows a snapshot of the running simulation. Experiments have been conducted using three different variables of dynamism δ for the environment. The first two columns in Tab.3 shows the $cost.ratio$ respectively for the BDI agent and the BDI^{rel}agent operating in a static environment ($\delta = 3$, a tree changes its location every 3 seasons, 45 rounds). Both agents attain an average of 43.375 eaten fruits on their

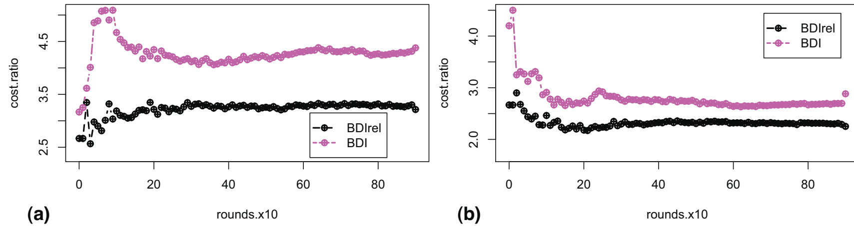


Fig. 2. Cost ratio in environments characterized by different dynamism: δ=1 (a) and δ=2 (b).

trials. However they have to pay quite different costs of belief update (BDI performs an average of 92.125 update operations, while BDI^{rel}78.437). Considering the $cost.ratio$, once agents have overcome their transitory phase they spend respectively 2.381 (BDI) and 2.012 (BDI^{rel}) costs for each eaten fruit.

The central columns show the performances of the two agents in environments with medium dynamism ($\delta = 2$, a tree changes its location every 2 seasons, 30 rounds). Here, due to the frequent tree changes, both agents rely on a less accurate knowledge model. In terms of eaten fruits the BDI agent attains a higher performances (38.185), clearly outperforming the BDI^{rel}agent (37.125). On the other hand, as far as costs of belief update are concerned, BDI^{rel}performs better: in fact, under the same conditions, BDI^{rel}does less belief update operations (85.312 vs. 101.437). As a consequence, BDI^{rel}considerably shows a better effectiveness with respect to the $cost.ratio$ (see Fig.2.b) whose value converges, at the end of the experiments, at an average of 2.252 belief update operations per achieved goal (vs. 2.882 for the BDI agent).

The last two columns shows the results in a highly dynamic environment ($\delta = 1$, a tree changes its location every season change, 15 rounds). Here, due to the massive epistemic activities, the standard BDI agent is able to maintain a more consistent and complete knowledge of the environment. Therefore, he performs better than the BDI^{rel}agent in terms of achieved goals (with an average of 33.937 number of eaten fruits against 33.875). On the contrary, BDI agent has to pay higher costs related to his epistemic activities, even beyond the initial transitory phase. In this case, the costs of belief update are 143.125 for the BDI agent and 107.875 for the BDI^{rel}agent. As depicted in Fig.2.a, the $cost.ratio$ reflects this difference by converging to a value of about 4.379 for the BDI and of 3.214 for the BDI^{rel}.

The experiments show a noticeable effect of the relevance-based filter of belief update on the BDI^{rel}performance. By reflecting agent effectiveness both in terms of goal achieved and in terms of costs for belief update, $cost.ratio$ is, indeed, a good indicator for analyzing the trade-off in agents performances (see Fig.2). On the one side the BDI agent is always the best in terms of goal effectiveness. BDI agents are *passively* affected by all incoming information, hence they obtain a precise knowledge of their surroundings. Consequently, the more an agent spends his resources for belief change, the more his beliefs will be correct and adequate with regard to the current state, and the more the agent will find fruits in the grid. On the other side, BDI^{rel}agents adopt an active perception style and exploits their relevance-based filter: if the incoming in-

put is not relevant, then the BDI^{rel}agent simply ignores it. This allows the BDI^{rel}agent to avoid wasting computational resources for doing belief update, thus resulting in a better global performance in terms of $cost.ratio$. Moreover, the results also show that the higher is the dynamism of the environment, the higher are the computational costs paid for belief update. In these conditions, BDI^{rel}agents have a concrete advantage in filtering noise and in considering only what is expected to be useful for achieving their current goals.

7 Conclusion and Related Works

We have presented in this work a mechanism of relevance-based belief update and implemented it in a BDI agent. Despite the simple scenario adopted, we think that our model can be applied straightforwardly to more complex scenarios and that our experimental results can be easily generalized. As the experimental results show, the costs for belief update are effectively reduced by using the mechanism for filtering relevant data.

It is worth noting that the notion of relevance is not new in the literature, as it has been extensively investigated in several domains like AI, philosophy and cognitive sciences. Most authors have been interested in a kind of *causal* (or *informational*) relevance based on various forms of conditional in/dependence. According to [9, 14], for instance, the concept of relevance coincides with the probabilistic concept of conditional dependence, in particular irrelevance is identified with conditional independence, and relevance is identified with the negation of irrelevance. There are some computational systems in the literature, inspired by Information Theory [15], which conceive relevance as a quantitative notion of informativeness. Low-level mechanisms have been proposed to model adaptive perception (see [18] for instance) and to relate relevance to action selection through classifier systems [17]. Moreover, few programming models exist explaining the relationship between perceptive processes and agent reasoning. Among others, Pereira & Tettamanzi recently proposed a formal model of goal generation where both relevance and trustworthiness of information sources are involved in rational goal selection [5], while few works face with agent models for appraising incoming input on the basis of cognitive relevance [12, 8]

Differently from the previous works, which are mostly interested in a notion of informational relevance, we have investigated in this work a practical notion of relevance. Our notion of relevance is indeed closer to the concept of relevance discussed in the psychological literature on motivation and emotion [10], where relevance is related to the subjective appraisal of a certain event, situation, object with respect to an agent's goals and intentions. We can refer to it as *pragmatic relevance* (or *goal relevance*) in order to distinguish it from causal (or informational) relevance (see also [6] for a discussion on the distinction between causal relevance and pragmatic relevance). Pragmatic relevance should be conceived in terms of perceived utility or beneficiality of an information with respect to an agent's intentions and goals. We think that this notion of pragmatic relevance is a crucial concept for the design of agent-based technologies that have to perform in highly dynamic and information-rich environments (e.g., the Web).

Directions for future works are manifold. We are actually working on a generalization of our model of pragmatic relevance which consists in adding a quantitative

dimension for intentions (i.e. the utility/importance of the intended outcome). In this generalized model, the degree of relevance of a certain input with respect to an agent's intention will also depend on the utility/importance of the intended outcome. Besides, we will consider in the future the relationships between our concept of relevance and intention reconsideration. Since the persistence of an intention over time depends on the persistence of those beliefs which support this intention (i.e. beliefs are reasons for intending), we will study how the relevance-based filter of belief update discussed in this paper may affect the persistence of intentions in an indirect way. Finally, we intend to develop in the future a more advanced model extending an agent's capability to manage his belief base (i.e., salience maps, dynamic Bayesian networks, influence diagrams, etc.).

References

1. R. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. John Wiley & Sons, Ltd, 2007.
2. M. Bratman. *Intentions, plans, and practical reason*. Harvard University Press, 1987.
3. R. Casati and E. Pasquinelli. How can you be surprised? the case for volatile expectations. *Phenomenology and the Cognitive Sciences*, 6(1-2):171–183, 2006.
4. C. Cherniak. *Minimal rationality*. MIT Press, Cambridge, 1986.
5. C. da Costa Pereira and A. G. B. Tettamanzi. Goal generation with relevant and trusted beliefs. In *Proc. of the Seventh International Conference on Autonomous agents and Multiagent Systems (AAMAS'08)*, pages 397–404. ACM Press, 2008.
6. L. Floridi. Understanding epistemic relevance. *Erkenntnis*, 69:69–92, 2008.
7. L. P. Kaelbling and S. J. Rosenschein. Action and planning in embedded agents. In P. Maes, editor, *Designing autonomous agents*, pages 35–48. MIT Press, Cambridge, 1990.
8. A. Koster, F. Koch, L. Sonenberg, and F. Dignum. Augmenting BDI with Relevance: Supporting Agent-based, Pervasive Applications. In *Mobile Interaction Devices (PERMID 2008) Workshop at Pervasive 2008, Sydney.*, 2008.
9. G. Lakemeyer. Relevance from an epistemic perspective. *Artificial Intelligence*, 97(1-2):137–167, 2004.
10. R. S. Lazarus. *Emotion and adaptation*. Oxford University Press, New York, 1991.
11. E. Lorini and C. Castelfranchi. The unexpected aspects of surprise. *International Journal of Pattern Recognition and Artificial Intelligence*, 20(6):817–833, 2006.
12. E. Lorini and M. Piunti. The Benefits of Surprise in Dynamic Environments: From Theory to Practice. In *Affective Computing Intelligent Interactions (ACII-07), LNCS Vol. 4738. Springer.*, 2007.
13. A. Omicini, A. Ricci, and M. Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17 (3), 2008.
14. J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufman, Cambridge, 1988.
15. C.E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:623–656, 1948.
16. H. Simon. *Models of thought*, volume 1. Yale University Press, 1979.
17. G. Weiß. Learning the goal relevance of actions in classifier systems. In *Proc. of the Tenth European Conference on Artificial intelligence (ECAI'92)*, pages 430–434, 1992.
18. D. Weyns, E. Steegmans, and T. Holvoet. Model for active perception in situated multi-agent systems. *Special Issue of Journal on Applied Artificial Intelligence*, 18:200–4, 2003.
19. M. Wooldridge. *An Introduction to Multiagent Systems*. John Wiley & Sons, Chichester, 2002.

Modularity and compositionality in Jason

Neil Madden and Brian Logan

School of Computer Science
University of Nottingham, UK.
{nem,bsl}@cs.nott.ac.uk

Abstract. In this paper, we present our experiences using the *Jason* agent-oriented programming language to develop a complex multi-agent application. We highlight a number of shortcomings in the current design of the language when building complex agents, and propose revisions to the language to allow the development of modular programs that facilitate code reuse and independent development. In particular, we propose a mechanism for modular construction of agents from functionally encapsulated components, and discuss alterations to the belief base language to enable more robust software engineering.

1 Introduction

Agent-oriented programming languages, such as *Jason* [1] and 2APL [2], have been the subject of a great deal of research and development in recent years. Building on the foundations of logic programming and theories of agency—in particular, the BDI (belief-desire-intention) model—they aim to raise the level of abstraction used in constructing complex modern software applications. However, while they have been successfully applied in a number of interesting problem domains, the literature contains relatively few reports of attempts to apply such languages to large-scale software development efforts.

In this paper we present our experiences of applying the agent-oriented programming language *Jason* to the development of a large-scale multi-agent system consisting of (relatively) complex *witness narrator agents* which report on events occurring in online persistent game environments [3]. Witness-narrator agents are embodied in a virtual environment and observe and narrate activities occuring within that environment. The deployed system consisted of a team of 100 agents which reported on events in a medium-scale persistent virtual environment over a period of several weeks. The agents had to handle a number of complex tasks during this period, including activity recognition, multi-agent coordination, generation of prose stories describing activities, and interaction with human participants. The system makes use of a range of technologies, including ontological reasoning, plan and activity recognition, and multi-agent coordination and teamwork. The architecture of the agents is organised as a collection of functionally encapsulated 'capability' modules to handle distinct tasks, such as low-level activity recognition, editing of reports from multiple agents, and generation of prose text for a particular output medium.

Our experiences with *Jason* indicate that it is a useful and flexible language which provides a clean, high-level approach to defining complex agent logic. However, we

also found the language to be lacking in some respects, particularly in relation to the development of more complex agents. In this paper we describe the problems that we encountered, and propose revisions to the language to allow the development of modular programs that facilitate code reuse and independent development. In particular, we propose a mechanism for modular construction of agents from functionally encapsulated components, and discuss alterations to the both the belief base language and plan execution to enable more robust software engineering.

The remainder of the paper is organised as follows. In section 2 we first give a brief introduction to the *Jason* programming language. We then discuss two related areas where we experienced some problems with the current design of *Jason*, describing the problems, possible solutions, and a concrete proposal for improvement. In section 4 we look at the choice of Prolog for the default belief base language in *Jason*, discuss some problems that this presents for modular agent construction, and describe an alternative based on Datalog. In section 5 we then look at the requirements for general modular construction of agents, and propose a simple module system that addresses these requirements. Lastly, we conclude with a look at related work (section 6) and some general conclusions in section 7.

2 Jason

Jason [1] is a Java-based interpreter for an extended version of AgentSpeak(L). AgentSpeak(L) is a high-level agent-oriented programming language [4] which incorporates ideas from the BDI (belief-desire-intention) model of agency. The language is loosely based on the logic programming paradigm, exemplified by Prolog, but with an operational semantics based on plan execution in response to events and beliefs rather than SLD as in Prolog. *Jason* extends this base language with support for more complex beliefs, default and strong negation, and arbitrary internal actions implemented in Java. The belief base of AgentSpeak(L) consists simply of a set of ground literals, whereas *Jason* supports a sizeable subset of Prolog for the belief base, including universally-quantified rules (Horn clauses).

Agents in *Jason*, as in other agent-oriented environments, are autonomous encapsulated processes that communicate with each other by sending messages (speech acts). Figure 1 shows a simple example *Jason* agent program which implements a bank account agent. The current balance of the account is stored as a ground fact in the belief base, along with a single Prolog-style rule for checking whether the account has sufficient funds for a withdrawal. Deposit and withdrawal functionality is implemented using three plans which inform the account holder whether a given withdrawal was possible and update the account balance accordingly. The syntax of *Jason* plans essentially consists of a single triggering event (such as a goal or belief addition or deletion, or a percept), a belief context pattern, and then a sequence of actions to perform if the plan is selected.[1] During execution, *Jason* first processes any events and updates the belief base. The interpreter then selects a single event to process and matches it against the plan library to select one or more plans to handle the event. Of these plans, a single plan

[1] In the interests of brevity, we have slightly simplified the presentation of *Jason* syntax and semantics.

```
/* Initial balance */
balance(0).
/* Check if withdrawal is permitted */
sufficient_funds(Amount,NewBalance) :-
    balance(Balance) &
    Balance >= Amount &
    NewBalance = Balance - Amount.

@deposit[atomic]
+!deposit(Amount)[source(AccountHolder)] :
    balance(Balance)
<-
    .print("Deposit ",Amount," from ",AccountHolder);
    -+balance(Balance + Amount).

@withdraw_overdrawn[atomic]
+!withdraw(Amount)[source(AccountHolder)] :
    not sufficient_funds(Amount,_)
<-
    .print("Attempt to go overdrawn from ",AccountHolder);
    .send(AccountHolder,tell,overdrawn).

@withdraw_ok[atomic]
+!withdraw(Amount)[source(AccountHolder)] :
    sufficient_funds(Amount,NewBalance)
<-
    .print("Withdraw ",Amount," by ",AccountHolder);
    -+balance(NewBalance);
    .send(AccountHolder,tell,withdrawn(NewBalance)).
```

Fig. 1. Example *Jason* program implementing a simple bank account.

is then selected to become an intention. Finally, one of the currently active intentions is selected and allowed to perform an action, before the cycle repeats. The complete cycle is shown in Fig. 2, adapted from [1], Chap. 4. Marking a plan as `atomic` (as in Fig. 1) ensures that all of its actions are run to completion before another intention is selected.

The process of constructing software using *Jason* proceeds at two levels. At a high-level, the problem domain is broken down in terms of a society of autonomous and cooperating (or competing) agents. In the BDI paradigm, programming in the large thus involves decomposition of the system into entities (agents) to which belief and other propositional attitudes can most naturally be ascribed. At a lower level, individual agents are authored in terms of their beliefs and goals, and plans which specify how to achieve the agent's goals and how to react to events. The primary mechanism for

1. Perceive the environment;
2. Update the belief base;
3. Receive communication from other agents;
4. Select socially acceptable messages;
5. Select an event;
6. Retrieve relevant plans;
7. Determine applicable plans;
8. Select one applicable plan;
9. Select an intention for execution;
10. Execute an action.

Fig. 2. *Jason* interpreter cycle.

structuring a *Jason* program in the small is therefore the plan.[2] Issues arise when a natural decomposition of the system into (intentional) agents results in entities with large numbers of plans. In such cases, a modular approach to agent development is often desirable.

3 Problems of Large-Scale Agent Programming

An agent-oriented approach to constructing large software has several advantages, such as encouraging separation of concerns, so-called loose coupling between components, and extending relatively naturally to a distributed environment in which messages are sent over a network to other remote agents. *Jason* provides good support for constructing multi-agent systems at this level, providing natural and easy to use speech-act based communications, and abstracting away from many of the details of the underlying infrastructure. At the level of constructing an individual agent, the BDI model of *Jason* and the sophisticated plan and belief base facilities it provides allow the developer to express complex logic in a concise and clear fashion. However, our experience with using *Jason* to develop a large and complex application indicates that there is a gap between these two levels that becomes more apparent as individual agents grow in complexity. In particular, *Jason* lacks any mechanism for decomposing an individual agent into constituent components or modules. Figure 3 shows the overall architecture of one of the agents in the system we developed, known as a *witness-narrator agent*. The agents we developed have a quite complex internal structure, consisting of a number of different competences ('capabilities') that are conceptually independent of one another to a large degree and communicate only through clearly defined interfaces. Each capability module consists of a set of *Jason* plans, along with some beliefs and rules, that are used to implement that particular competence. For instance, the reporting module contains plans for detecting and recording activity occurring within an environment, while the presenting module has plans and rules for formatting a report for a particular output medium (HTML, Atom). A separate coordination module handles communication and

[2] Although traditional Prolog-style rules in the belief base can also form a significant part of the codebase.

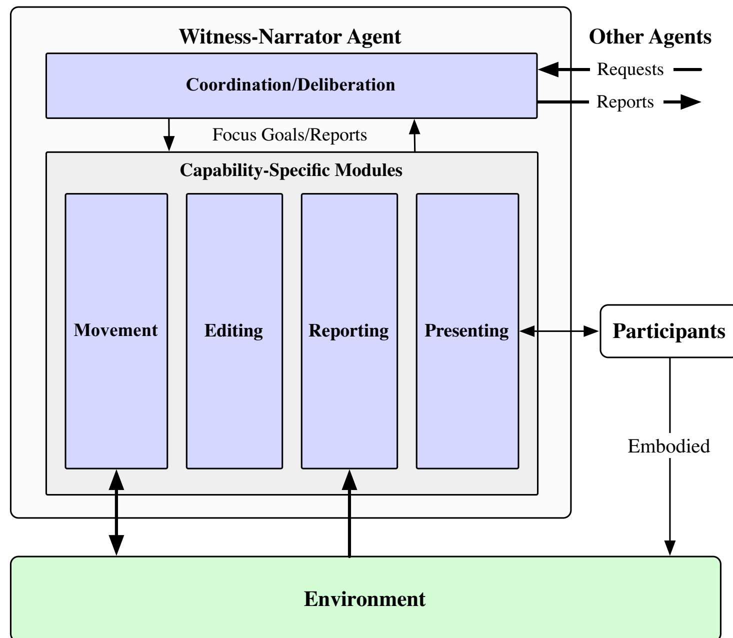


Fig. 3. Architecture of witness-narrator agent.

interaction with other agents, such as team formation. The current implementation of witness-narrator agents in *Jason* implements such modules simply as a set of files which are included one at a time into a main agent program. There are a number of drawbacks with this approach:

– there is the possibility of name clashes between belief and goal relations defined in separate modules;
– it may be desirable for plans in different modules to react to the same triggering event;
– the order in which modules are included can have surprising effects on the execution of the agent.

The first problem is one of namespace management, and can be addressed to a certain degree by adopting coding guidelines to ensure that beliefs and goals in separate modules have different names, for instance using a simple unique prefix for each module. However, this issue becomes more important when we consider third-party modules that are developed independently by different organisations. While guidelines can be adopted to try to ensure that unique prefixes are chosen (such as incorporating the institution name into the prefix), such approaches can be cumbersome to use. The possibility that agents may dynamically locate and acquire new modules at runtime (e.g., OWL ontologies) also suggests that good namespace management should be built in to the language itself. This could be achieved in a straightforward way by using a mechanism

similar to that adopted for XML [5], in which components of an agent are associated with an unique Uniform Resource Identifier (URI), and in which short string prefixes can be assigned to these URIs within a source file. Such a scheme presents a good combination of flexibility, safety, and ease of use.

The second and third drawbacks present more serious problems. The semantics of plan execution in *Jason* require a single plan to be selected to respond to any particular triggering event. This prevents multiple modules from responding to the same event in different ways, and introduces a form of implicit coupling between modules that must be explicitly resolved by the agent developer. For instance, in the development of the witness-narrator agents, certain events were of interest to both the reporting and general movement modules. If an agent is attacked, for example, this is both a cause for the movement module to take action (to evade the attack), but also presents a potentially interesting event that the reporting module may want to record. In the current implementation, this is achieved by having one module handle the event, and then to generate a secondary event purely for the other module to be notified. This requires both explicit cooperation between the modules (and corresponding effort from the developer), and also introduces extra code whose purpose can be obscure to a reader of the program.

The third problem stems partly from the second, in that if two modules attempt to react to the same triggering event (and the developer hasn't noticed this), then which plan gets to run depends on the plan selection function, which by default is based on the order in which plans are defined. This can lead to some surprising and difficult to understand behaviour as one plan appears not to be triggering correctly, and the actual cause of this is another plan in an entirely unrelated module. The problem also occurs if one module included earlier in the sequence includes a plan that effectively matches many or all triggering events, for instance if an unbound variable is used for the trigger. This then takes precedence over any subsequent plans introduced by later modules.

These problems can be seen as stemming from the overall problem of how to construct an agent by composing independent functional components. Ideally, an agent could be constructed by glueing together such independent reuseable modules. The most important property that such a module system should address is that the behaviour of an agent composed of separate modules should not depend on the order in which those modules are composed. In other words, we would like composition of modules to be both associative and commutative, so that the order and combination of modules does not effect the resulting behaviour. This property is particular important in an event-driven architecture such as *Jason*, where plan execution in response to events can sometimes be difficult to predict. Minimising effects caused by code refactoring helps to reduce the opportunities for confusion when constructing sophisticated software.

The requirements for associative and commutative composition of modules have implications for several areas in the design of *Jason*. Firstly, we must ensure that the composition of beliefs from all included modules has the same meaning, and produces the same runtime inference behaviour, no matter in what order those beliefs are added to the belief base of the agent. This has an impact on the choice of belief representation language and reasoning strategy employed in the belief base. Similar considerations must be taken into account for goals. Secondly, in order to minimise conflicts between modules, it is important to support encapsulation of beliefs, goals and plans that are

internal implementation details of a module, while exposing those that form part of the interface. This includes preventing simple name clashes, but also more important issues of belief and goal scope and visibility.

4 Belief Base Compositionality

One of the enhancements of *Jason* over AgentSpeak(L) is the support for a substantial subset of Prolog in the belief base language, including backward-chaining rules. This considerably increases the power of the language, and allows for succinct descriptions of some problems, while also allowing the *Jason* developer to take advantage of the large number of resources available for existing Prolog developers. However, despite these advantages, it is not clear whether Prolog is in fact the most appropriate choice for a belief language.

- As already noted, the order in which clauses are added to the belief base has an explicit effect on execution in Prolog (and hence *Jason*), with clauses defined earlier in a program having precedence over later clauses. In some cases, reversing the order of two clauses can lead to incorrect behaviour or even nontermination of a previously correct definition. This clearly violates the requirements for compositionality of beliefs.
- The backtracking execution strategy of Prolog may not be the most efficient way of handling large numbers of beliefs, particular when these may be stored in a relational database system, or other persistent store. For example, the agents used in our software used a MySQL database for persistent beliefs (archives of generated reports and previous activity), and over the several weeks the system was running, acquired many thousands of ground facts.
- Prolog is a computationally complete language, capable of expressing nonterminating algorithms. This is an undesireable property for a belief language, the purpose of which is largely to perform limited inferences to determine if a plan is currently applicable. The possibility that such a belief context check may in fact not terminate has consequences for the rest of the plan execution semantics, and indeed could entirely halt the agent or even the entire MAS depending on the implementation and infrastructure in use.

The sentivity to belief addition order is particularly important for agents in dynamic environments, where the agents are continually acquiring (and possibly revising) their beliefs over time. This is also a concern with regard to the modularity issues discussed in the previous section: if two modules add facts or rules to the same belief relation, we would like the order in which they are added to not affect the resulting behaviour. For example, the witness-narrator agents combined default rules for classifying observations together with dynamically acquired knowledge regarding individuals. For instance, an agent may have a default rule for determining whether an animal can fly, perhaps by reasoning about its anatomy or species, but then can also record instances where it has directly observed individuals flying. The default rule is likely to be defined when the agent is created, whereas the specific instances are added to the belief base as they are observed. In *Jason* this results in the observations being ordered after the

default rule, and due to the execution strategy of Prolog, this will lead to the inefficient and surprising behaviour that the agent will spend time trying to infer if a particular individual is capable of flight when it already has an explicit fact recording this information in its belief base. Another area where the use of Prolog caused some difficulties was in the translation of ontological rules from an OWL ontology into equivalent belief base rules. While a number of such translations have been described in the literature for Prolog-like rule languages (in particular, Datalog), the details of the translation for Prolog itself are complicated by the sensitivity to ordering of rules, and naïve translations can easily result in rules that do not terminate on certain inputs.

In the implementation of the witness-narrator agents, we worked around these problems by providing a custom belief base implementation which implemented slightly different semantics to that of Prolog, by always preferring ground facts to rules, regardless of the order in which they were added to the belief base. This was sufficient to ensure correct operation of the software in a wide range of cases, but the possibility for encountering a non-terminating query could not be entirely ruled out.

4.1 Datalog as a Belief Base Language

A better solution to the problem would be to replace Prolog with a more suitable knowledge representation language. A number of languages seem applicable, including restricted versions of Prolog designed for interfacing with relational databases, such as Datalog, or a description logic based language [6], such as OWL. Current description logic languages, however, are unable to express many interesting rules that are (easily) expressible in Datalog. In addition, restricting the belief base language to unary and binary predicates (concepts and roles, respectively) makes some problems rather clumsy to express, and can lead to difficulties keeping track of all the information associated with an individual, e.g., when performing belief revision.

Datalog offers many of the advantages of Prolog (the entire belief base used in our witness-narrator agent framework could have been expressed in Datalog with very few changes) while affording more efficient implementation. In particular, the properties of Datalog that make it suitable as a belief language include:

- the order of clauses in a Datalog program does not effect the semantics of query answering;
- the limitations on the language allow all queries to be answerable in polynomial time;[3]
- more efficient query answering, particularly for larger sets of beliefs.

As Datalog was designed to be a database query language, it should also provide better support for large belief bases, such as those backed by a persistent relational database management system. The non-recursive subset of Datalog has a natural translation into the relational algebra, allowing for efficient and direct compilation of Datalog belief rules into equivalent SQL queries or views. As a further benefit, there also exist translations from various description logics into Datalog. The use of Datalog as a belief base language would therefore go some way towards supporting efficient ontological reasoning in *Jason*, in particular supporting efficient ABox queries over large ontologies.

[3] Although this isn't true for various extensions to support negation or disjunctive heads in rules.

5 Encapsulating Beliefs, Goals and Plans

The problems described in section 3 indicate that some mechanism for modular decomposition of individual agents is needed in *Jason*. Such a mechanism could take a number of forms, ranging from a relatively simple module or namespace mechanism, up to a complex object-oriented solution, complete with component instantiation, inheritance, and encapsulation. It could be argued that the agent abstraction could also be used at this level: an individual agent in a society being itself composed of a society of simpler agents. Such an approach is intuitively appealing, but it is not clear whether the advantages of agent-oriented programming in the large also hold for development of agents themselves. In addition, the capabilities within our system do not naturally fit with the usual notion of an 'agent', and it seems unnatural to try and shoe-horn the architecture into layered societies of agents. In this section we examine the requirements and desireable features of a modular approach to building agents, and tie these to the specific agent-oriented programming language, *Jason*.

For *Jason*, the requirements for a module system are relatively modest, as the agent level already provides many of the more sophisticated features, such as dynamic instantiation of agents and communication interfaces. From our experience, the main requirements for a module system in *Jason* would be as follows:

1. to simplify the reuse of software components by allowing functional units of code, including beliefs, goals, and plans, to be encapsulated into independent modules;
2. to provide a mechanism for avoiding name clashes between goals and beliefs from separate modules;
3. to allow each module to respond independently to events;
4. to provide a mechanism for controlling which beliefs and events are visible to other modules, and which are private to a particular module;
5. to ensure, as far as possible, that the order in which modules are included in an agent has no effect on the resulting behaviour of the agent.

These requirements are sufficient to address the immediate problems that were experienced during construction of the witness-narrator agent system. We do not consider more advanced functionality, such as parameterisation of modules, instantiation of multiple instances of a particular component, or customisation of plan selection or intention execution functions.

5.1 A Module System for *Jason*

Based on the requirements outlined above, we sketch a proposal for a module system for *Jason*. A *Jason module* consists of the following elements:

1. a local belief base, containing any beliefs that are private to the module;
2. a local goal base;
3. a plan library, implementing the functionality of the module;
4. a local event queue, for belief and goal update events that are local to the module;
5. a list of exported belief and goal predicates;
6. a list of imported modules;

7. a unique identifier (URI) that acts as a prefix for all belief and goal symbols in the module, based on XML namespaces;
8. a mapping from simple string prefixes to imported module URIs.

A module is therefore a subset of the functionality of an agent, encapsulated into a functional unit, along with an URI for identification. An agent is then defined as a composition of modules, together with an interpreter based on the original *Jason* BDI interpreter. Composition of modules within an agent is a flat one-level hierarchy, with the agent as the root. Nested sub-modules are not permitted in this scheme. This approach greatly simplifies the treatment of beliefs and events, while still addressing all of the requirements that we have described. This implies that there is only a single copy of each module within each agent, and references to that module are shared between all other modules that import it.

We adopt the XML approach to namespaces, where each module is considered as a separate namespace, identified by a URI. To ease the use of such a system, we also adopt the XML mechanism of allowing each module to declare a set of simple string prefixes that expand into the full URI reference for another module. Thus the belief predicate symbol `foo:fatherOf` would expand by looking up the prefix 'foo' in the current module's prefix mapping and substituting it for the corresponding URI, e.g., into `http://example.com/foo#fatherOf`. This mechanism could be layered on top of *Jason*'s existing annotation mechanism, so that the belief actually expands into a literal like `fatherOf(...)[module("http://example.com/foo")]`. Goal symbols are also scoped in an identical fashion, such that a goal like `!nwn:travel(Destination)` is expanded into `!travel(Destination)[module("http://example.com/nwn")]`. The URI used for a module, as well as the prefix mappings used by that module, could be defined in a *Jason* source file using simple directives, e.g.,:

```
{ module("family", "http://example.com/family.asl");
  import("friend", "http://example.com/friend.asl");
  export(["fatherOf/2","brotherOf/2","!birthday/1"]); }
```

This example syntax declares that the source file implements a module that is identified by the URI `http://example.com/family.asl`, and which imports a 'friend' module with a similar URI. Both modules are given simple string names that can be used within the source file in place of the full URIs (here, the strings 'family' and 'friend' are used respectively). Note that the string prefixes are just a convenience within this source file, and only the URIs are significant outside of the module. The `import` directive would also ensure that the specified module is loaded (once) into the agent, if it has not already been loaded[4]. Finally, an `export` directive specifies which belief and goal predicates should be exported from this module, as discussed below.

Each module is considered to contain its own independent belief base. By default, any belief additions or revisions performed by a module are applied to its own local belief base. A mechanism is provided to override this behaviour, by allowing some beliefs (and goals) to be declared as *exported*. What this means is that these beliefs (and

[4] We do not specify how module URIs are used to locate implementations.

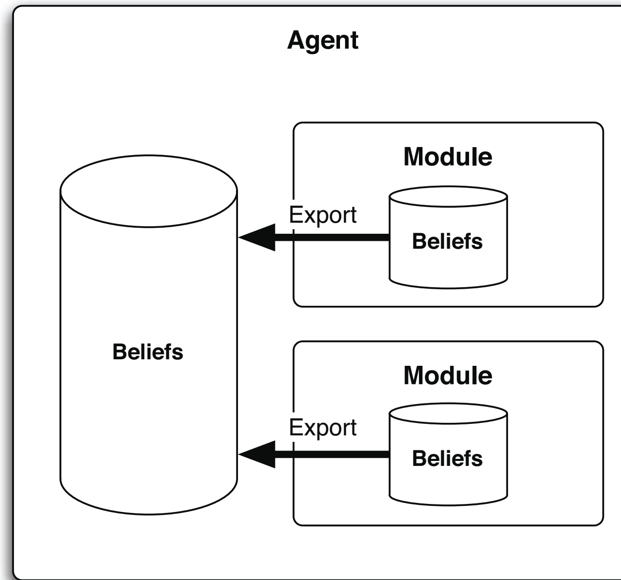


Fig. 4. Scoping of beliefs within modules with export.

goals) are not added to the module's own belief base, but are instead delegated to the belief base of the agent itself, as shown in Fig. 4. The same approach is used for goals, in that each module has a private goal base, while exported goal symbols are added to the main agent's goals. This partitioning of beliefs and goals into module-private areas also has implications for the generation and propagation of events arising from belief and goal addition and deletion (so-called 'internal events'). A belief or goal predicate that is considered to be private to a module can be seen as an implementation detail of that module, and so events relating to beliefs or goals matching that predicate should not be visible outside of that module. Otherwise, other modules might become reliant on these events, and therefore subject to errors if those implementation details are changed in future updates to a module. However, it is clearly desireable for a module to be able to respond to events arising from changes in its own internal state. We therefore also include a mechanism for scoping of events on a per-module basis. Each module has a private event queue. Events arising from changes to beliefs or goals in a per-module belief or goal base are added only to that module's event queue. Events arising from changes to the agent's main belief or goal base (i.e., from exported beliefs or goals) are added to the agent's main event queue, as in the current *Jason* implementation.

5.2 Interpreter Cycle Changes

The changes we have described to incorporate modules into *Jason* imply a number of changes to the *Jason* BDI interpreter cycle, shown previously in Fig. 2. We here

describe the changes to the interpreter cycle that are required for our proposal. Items without comments are assumed to be identical to the current *Jason* implementation.

1. *Perceive the environment.*
2. *Update the belief base.* Belief updates must now take into account the scoping of beliefs. This is achieved by examining a belief update for any URI prefix and using this to determine which module's belief base should be updated (taking into account export lists). Any unqualified belief updates are performed using the agent's main belief base.
3. *Receive communication from other agents.*
4. *Select socially acceptable messages.*
5. *Select an event.* The set of events to choose from is taken as the union of all of the pending event queues for each module and the agent's main event queue.
6. *Retrieve relevant plans.* The set of potentially relevant plans depends on the scope of the triggering event. For events that are local to a module, then only that module's plan library should be considered. For events scoped at the level of the overall agent, the set of potentially relevant plans is the union of the plan libraries of all modules. Determining whether a plan matches a triggering event is done by first expanding any URI references in both into *Jason* annotations, as described previously, and then performing matching as in the current implementation.
7. *Determine applicable plans.* As for relevant plans, determining applicable plans (i.e., those whose belief context is satisfied), must also respect scoping of those beliefs and expanding URI references.
8. *Select one applicable plan.* As described in Sec. 3, one of the motivating justifications for this work, and a key compositionality requirements, is that each module should be able to respond independently to events. It therefore seems sensible to extend this rule in the cycle so that up to one applicable plan is selected *per module*. However, this approach is more complicated than it immediately appears. The problem is that belief and goal events resulting from an existing intention cause any responding plans to become part of that same intention structure. Clearly this presents a problem if multiple plans are selected, as only one can become part of the intention (without much more drastic changes to the structure of intentions). We discuss how to tackle this problem below.
9. *Select an intention for execution.*
10. *Execute an action.*

As described, the problem of allowing multiple modules to respond to the same event is more complex than it initially appears, due to interactions with the BDI model of *Jason*. In determining a solution to this problem, it is worth considering the types of events that can occur, and what an appropriate behaviour should be in each case. *Jason* currently distinguishes between so-called 'internal' events, arising from changes to the agent's internal state (beliefs and goals) caused by executing intentions, and 'external' events, caused by the arrival of new percepts and messages from other agents. External events always cause a new intention structure to be created, whereas internal events reuse the intention structure that generated the event. One solution, then, would be to only allow multiple modules to react to external events (creating a new intention for each module), and treat internal events as before, in which case only one module would

get to respond. However, this seems overly restrictive, as it is reasonable for multiple modules to respond to *belief* updates caused by an executing intention, whereas a *goal* update should only result in one course of action being taken, to avoid conflicting or incoherent behaviour. Belief change events, however, are largely *incidental* rather than intentional, and therefore should not put such strong constraints on resulting behaviour. For instance, an agent may want to perform various house-keeping tasks in response to a belief change: inferring further conclusions (forward-chaining), informing other agents of the change, and so on, but this is much less likely for goals. It therefore seems more sensible to distinguish events not by an internal/external distinction, but rather by a belief/goal distinction. We therefore propose that plans responding to belief update events should always create new intention structures, allowing multiple modules to respond to the same event, whereas as goal update events should follow the existing behaviour in *Jason*: only a single plan can respond to the event and this plan becomes part of the same intention that caused the event (if one exists). This addresses the compositionality requirements for beliefs, but still requires some coordination between module authors wishing to respond to the same goal event. In practice, this addresses all of the issues that we encountered in the witness-narrator agent framework.

6 Related Work

Most popular current programming languages support some form of module system allowing for decomposition of large programs into functionally encapsulated components. Approaches range from simple namespace mechanisms, to sophisticated module systems supporting parameterisation, instantiation, and complex nested module structures. Within the realm of agent-oriented programming languages and frameworks, a number of proposals have been presented in the literature. The notion of modules as 'capabilities' was developed within the context of the JACK agent framework [7] and has been extended in the JADEX framework [8]. A *capability* in this proposal is a collection of plans, beliefs, and events together with some scoping rules to determine which of these elements are visible outside of the module, and which are encapsulated. Like the proposal described in the current paper, capabilities represent a middle layer between that of an agent and the level of individual plans and beliefs. Capabilities address the concerns of avoiding name clashes and hiding of implementation details, while also supporting multiple instances of the same module to be created. The later work with JADEX extends the concept to include a more flexible notion of scoping for beliefs and events, as well as allowing capabilities to be parameterised and dynamically instantiated. However, capabilities do not address the problems of plan selection and execution that we have described, i.e., to allow plans from different modules to each have a chance to react to an event. A proposal for incorporating a similar notion of modules has been described for an extended version of the 2APL programming language [9]. As with capabilities, extended 2APL modules can be instantiated multiple times, or can be declared as *singleton* modules, for which a single instance of the module is shared within an agent. 2APL modules can contain any elements that an agent can contain, including plans, beliefs, goals, and action specifications, and are similar to agents in many respects. Each module can be executed by the module (or agent) that instantiated it, and

can receive and generate belief updates, goal revisions, and other events. For *Jason*, a simple module system has been developed as part of the work on integrating ontological reasoning in the JASDL system [10]. However, this work concentrates only on allowing per-module customisation of the various plan and intention selection functions in *Jason*, and does not address the modularity concerns described in the current paper. The changes described in Sec. 5.2 to the processing of events are based on the original scheme used in the Procedural Reasoning System (PRS) [11] (Sec. 8.2), in which belief change events cause new intentions to be created, whereas goal changes reuse the current intention structure. Given AgentSpeak's (and therefore *Jason*'s) historical basis in the PRS architecture, it seems natural to revert to this scheme.

While many agent programming languages use Prolog as a belief base language, e.g., *Jason*, 2APL, GOAL, there has been some work on alternative belief representation languages. The use of alternative knowledge representation languages for the belief base of a *Jason* agent has been considered in the context of adding support for ontological reasoning and OWL [12, 10]. However, this work has concentrated on extending the Prolog facilities of *Jason* with support for ontological reasoning, rather than replacing the existing belief base language. In [13] an approach to abstracting an agent language from any particular knowledge representation format is presented. While the authors note that Datalog and SQL meet their requirements for a *Knowledge Representation Technology*, the focus of the paper is on translations between knowledge representation technologies rather than the practicalities of any specific technology. Most agent programming frameworks (including the implementation of *Jason*) allow customising or replacing the default belief base to a certain degree. While such customisation can facilitate the integration of 'legacy belief bases', there remains a need for for a practical knowledge representation formalism, even if only as a default, and we feel that that Datalog is a more appropriate choice than Prolog for a default belief-base language.

7 Summary

In this paper we identified a number of problems with the *Jason* agent-oriented programming language motivated by our experience of building a moderately large and complex piece of software using *Jason*. While *Jason* provides a clean and elegant framework for building sophisticated multi-agent systems, it provides less support for developing complex agents with diverse, interacting capabilties. We identified two key problems: a lack of support for modular software development, and an order-dependence in the semantics of the belief representation language which makes it hard to compose modules and to author plans within a module. To address these problems we proposed revisions to the language to simplify the construction of agents from functionally encapsulated components, and changes to the belief base language and plan execution to support more robust software engineering.

In future work, we plan to investigate further revisions to *Jason* suggested by our experiences developing the witness narrator agents system, including changes to the triggering conditions of plans and the semantic characterisation of percepts.

References

1. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons Ltd (2007)
2. Dastani, M.: 2APL: a practical agent programming language. Autonomous Agents and Multi-Agent Systems **16** (2008) 214–248
3. Madden, N., Logan, B.: Collaborative narrative generation in persistent virtual environments. In: Proceedings of the AAAI Fall Symposium on Intelligent Narrative Technologies, Arlington, Virginia, USA (November 2007)
4. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: MAAMAW '96: Proceedings of the 7th European workshop on modelling autonomous agents in a multi-agent world, Springer-Verlag (1996) 42–55
5. Bray, T., Hollander, D., Layman, A., Tobin, R.: Namespaces in XML 1.0, second edition. Technical report, W3C (2006) `http://www.w3.org/TR/2006/REC-xml-names-20060816`.
6. Baader, F., Calvanese, D., McGuiness, D., Nardi, D., Patel-Schneider, P., eds.: The Description Logic Handbook. Cambridge University Press (2003)
7. Busetta, P., Howden, N., Rönnquist, R., Hodgson, A.: Structuring BDI agents in functional clusters. In Jennings, N.R., Lespérance, Y., eds.: Intelligent Agents VI, LNAI 1757, Springer (2000) 277–289
8. Braubach, L., Pokahr, A., Lamersdorf, W.: Extending the capability concept for flexible BDI agent modularization. In: Proceedings of ProMAS 2005, LNAI 3862, Springer (2005) 139–155
9. Dastani, M., Mol, C.P., Steunebrink, B.R.: Modularity in agent programming languages: An illustration in extended 2APL. In: Proceedings of the 11th Pacific Rim International Conference on Multi-Agent Systems (PRIMA 2008), LNCS 5357, Springer (2008) 139–152
10. Klapiscak, T., Bordini, R.H.: JASDL: a practical programming approach combining agent and semantic web technologies. In: Proceedings of DALT 2008. (2008) 91–110
11. Myers, K.L.: Procedural reasoning system user's guide. Technical report, Artificial Intelligence Center, SRI International, Menlo Park, CA (1997)
12. Moreira, A.F., Vieira, R., Bordini, R.H., Hübner, J.: Agent-oriented programming with underlying ontological reasoning. In: Proceedings of the 3rd International Workshop on Declarative Agent Languages and Technologies (DALT), Utrecht, Netherlands (July 2005) 132–147
13. Dastani, M., Hindriks, K.V., Novák, P., Tinnemeier, N.A.M.: Combining multiple knowledge representation technologies into agent programming languages. In: Proceedings of DALT 2008. (2008)

A Formal Model for Artifact-Based Environments in MAS Programming

Alessandro Ricci and Mirko Viroli

DEIS, Alma Mater Studiorum – Università di Bologna
via Venezia 52, 47023 Cesena, Italy
{a.ricci,mirko.viroli}@unibo.it

Abstract. In this paper we provide a formalisation of the computational and programming model behind CArtAgO, a platform/infrastructure for programming and executing *artifact-based* computational environments in MAS. Such environments are realised as set of *workspaces* where agents work together by instantiating, sharing, co-using *artifacts*, i.e. first-class entities of the agent world representing resources and tools that agents can exploit so as to support their individual and collective activities. Besides rigorously describing the main features of the artifact abstraction and the interaction model based on agent use and observation of artifacts, the formalisation aims at fostering the integration of CArtAgO with existing agent programming languages, in particular with those with a well-defined formal model and semantics.

1 Introduction

Environment programming in Multi-Agent Systems (MAS) accounts for considering the computational environment where agents are situated as a *first-class abstraction*, to be suitably designed and *programmed* so as to improve MAS engineering [13]. The background idea is that the environment can be an effective place where to encapsulate functionalities for MAS, in particular those that concern the management of agent interactions and coordination. This turns out to be useful for defining and enacting into the environment strategies for MAS coordination, organisation, and security—the interested reader can find in [16] a survey of works endorsing this viewpoint[1].

In this context, CArtAgO (Common Artifact infrastructure for Agent Open environment) [14] has been proposed as a general-purpose platform/infrastructure for building shared computational worlds – referred to as *work environments* – that agents, possibly belonging to heterogeneous agent platforms and written using different agent programming languages, can exploit to work together inside the MAS. Being based on the A&A (Agents and Artifacts) meta-model [10], CArtAgO's work environments are modelled as set of distributed *workspaces*, containing dynamic sets of *artifacts* (see Fig. 1). From the agent viewpoint, artifacts are first-class entities of agents' world, and represent resources and tools that agents can dynamically instantiate, share and use to support individual and collective activities. From the MAS designer viewpoint, artifacts are useful to uniformly design and program those abstractions inside a MAS that are not suitably modelled as agents, and that encapsulate functions to

[1] This sums up the the results of three years of E4MAS (Environment for Multi-Agent Systems) workshop, held at AAMAS from 2004 to 2006

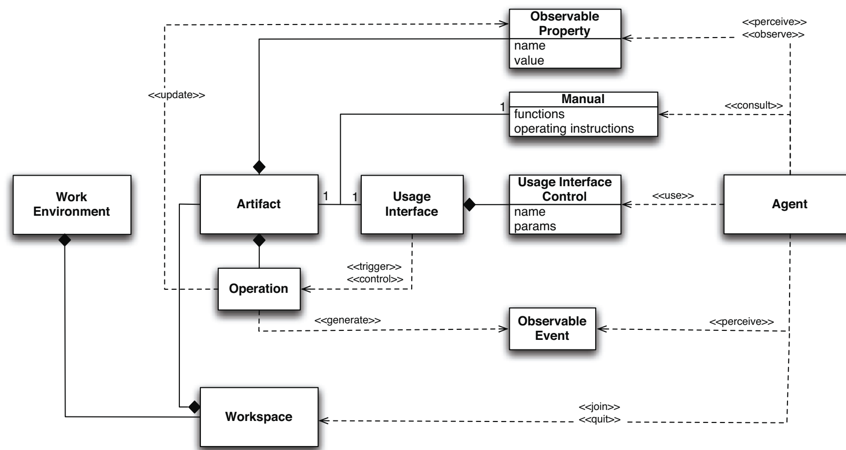


Fig. 1. A&A meta-model expressed in the Unified Modelling Language (UML).

be exploited by individual agents or the overall MAS—for instance mediating and empowering agent interaction and coordination, or wrapping external resources. So, if agents are the basic bricks to design the autonomous and pro-active part of the MAS, artifacts are the basic entities to organise the *non-autonomous*, *function-oriented* part of it—the latter being exploited and controlled by the former to achieve the MAS goals. For this reason, instead of being characterised in terms of goals, tasks or autonomous actions like in the case of agents, artifacts are characterised by the set of functionalities (*operations*) that they provide to their user, and a *usage interface* for agents to exploit such functionalities. Differently from the interaction models introduced for inter-agent communication, typically based on some form of high-level ACL and speech-act-like theories, the interaction model ruling agent-artifact interaction is based on *use* and *observation*: agents use artifacts by acting on their usage interface so as to trigger and *control* the execution of the operations, and by perceiving *observable events* generated by artifact and *observable properties* that constitute artifact observable state.

In this paper we provide a formalisation of CArtAgO computational model (Section 2), that is – more generally – a formalisation of the notion of artifact and workspaces as defined by the A&A conceptual model. The motivation and objective of the formal model is twofold: (i) making the technical aspects of CArtAgO non-ambiguous, and avoiding underspecification so as to enhance their understanding; (ii) fosters the integration of CArtAgO with existing agent programming languages, in particular with those with a well-defined formal model and semantics.

To authors' knowledge, this is the first formal model in MAS programming literature providing a comprehensive and detailed account of concepts that typically are part of agent-environment interaction models, such as the use-of and the observation-of the environment, and related rich model for actions and perceptions. As a related work, we mention here the model of situated MAS proposed by Ferber and Müller [6] – introducing a general theory of actions for reactive agents situated in an environment with its own behaviour – and the

one developed by Weyns and colleagues [15], which models the environment as a level in a layered software architecture perspective of MAS, organised in functional blocks. Both works have been inspiring for the formal model presented in the paper: differently from these work, our approach is specifically targeted to define a general-purpose computational and programming model, introducing in particular a formalisation of the basic abstraction (the artifact abstraction, specifically) used to structure and modularise agent computational environments. Another related work to CArtAgO in general is GOLEM [2], which provides a declarative model to specify environments for cognitive agents based on the KGP model of agency [9], using the notion of *objects* and *containers* in a similar way w.r.t. artifacts and workspaces.

After presenting the model, in Section 3 we conclude the paper with a brief discussion pointing out some highlights about the model that we think are important for MAS programming and agent programming languages.

2 A Formal Model for Artifacts and Workspaces

We adopt a formalisation style which is standard and mostly compatible with existing models of agent programming languages (examples are AgentSpeak [12, 1], 3APL [7], 2APL [3]). Namely, we describe the operational semantics of CArtAgO by a transition system, which specifies how a multiagent system state – expressed in terms of a structured term – evolves into another by a single computation step, modelling either agent or artifact internal computations, or interactions between agents and artifacts. Furthermore, this formalisation style straightforwardly leads to an executable specification in frameworks such as the MAUDE term-rewriting language, which provide analysis tools like LTL model-checking [5].

We necessarily abstract away from some aspects that are orthogonal to CArtAgO: we *(i)* assume a simple agent execution cycle that is compatible with most existing agent programming platforms, *(ii)* accordingly abstract from details concerning agent mind structure and inner (reasoning) processes, *(iii)* model execution of artifact steps in terms of abstract automata, to foster exploitation of different concrete artifact languages as well. Also, for the sake of space and understanding, our formalisation also neglects some aspects like timeouts and linkability—however, the model has been conceived to be easily extended to include such features.

2.1 Structure

We adopt the following conventions, some of which are standard in the formalisation of programming languages and systems [8]. We introduce the following meta-variables[2]: g ranges over agent state, a over artifact state, τ over artifact templates, b over agent bodies, s over workspaces, and o over artifact triggered operations; furthermore, we introduce meta-variables for the unique identifiers of these constructs, $\gamma, \alpha, \tau, \beta, \sigma, \omega$, respectively. A meta-variable plays the role of non-terminal symbol in syntactic definitions, while in operational semantics will be used along with their variations (x, x', x_0 and so on) to range over elements of the corresponding syntax. Given meta-variable x, a set of elements of kind x is

[2] They are called meta-variables to avoid confusion with variables of the language, like artifact variables

mas	$::=$	$\langle \overline{g}, \overline{s} \rangle$	multiagent system
s	$::=$	$\langle \sigma, \overline{a}, \overline{b} \rangle$	workspace
g	$::=$	$\langle \gamma, step, mstate, perstate, prog, \overline{\sigma} \rangle$	agent
$step$	$::=$	$\mathsf{Perc} \mid \mathsf{UpdSt} \mid \mathsf{ChAct} \mid \mathsf{ExecAct}$	cycle step
$perstate$	$::=$	$\langle \overline{prop}, \overline{artevt}, \overline{actevt} \rangle$	percept state
$prop$	$::=$	$\langle \alpha, pname \mapsto pvalue \rangle$	observable property
$artevt$	$::=$	$\langle \alpha, evtvalue : evttype \rangle$	artifact(-related) event
$actevt$	$::=$	$\langle action, res, feedback \rangle$	action(-related) event
$prog$	$::$	$\langle nextf, actf \rangle$	agent program
$action$	$::=$	$\texttt{joinWsp}(\sigma) \mid \texttt{quitWsp}(\sigma) \mid \texttt{makeWsp}(\sigma)$	workspace actions
	$\mid$	$\texttt{use}(\alpha, oname, opars, sid_\perp, timeout) \mid \texttt{sense}(sid, filter, timeout)$	use/sense actions
	$\mid$	$\texttt{focus}(\alpha, sid_perp) \mid \texttt{stopFocus}(\alpha) \mid \texttt{observeProp}(\alpha, pname)$	observation actions
a	$::=$	$\langle \alpha, \tau, aname, \overline{uic}, \overline{prop}, instate \rangle$	artifact
uic	$::=$	$\langle [guard] oname \rangle$	usage interface control
$instate$	$::=$	$\langle wstate, \overline{var}, \overline{o} \rangle$	inner state
o	$::=$	$\langle \omega, oname, ostate, ocxt \rangle$	pending operation
t	$::=$	$\langle \tau, \overline{opdef}, \overline{prop}, \overline{var}, init, manual \rangle$	artifact template
$opdef$	$::=$	$\langle oname, ostate, onext, oguard \rangle$	operation definition
b	$::=$	$\langle \beta, \gamma, \overline{sns}, \overline{artevt}, \overline{actevt}, \overline{action}, \overline{otrig}, \overline{aobs} \rangle$	agent body
sns	$::=$	$\langle sid, \overline{artevt} \rangle$	sensor
$otrig$	$::=$	$\langle \omega, \alpha, sid_\perp \rangle$	triggered operation
$aobs$	$::=$	$\langle \alpha, sid_\perp \rangle$	observed artifacts

Fig. 2. Syntax of CArtAgO structures

ranged over by meta-variable $\overline{x}$—note that x and $\overline{x}$ are hence different meta-variables. Concatenation symbol ";" is used for disjoint union of sets or elements, symbol "$\setminus$" for set difference, and "$\emptyset$" is the empty set. Meta-variable $x_\perp$ is used to range over all elements of kind x plus $\perp$—typically meaning "no element" as when a sensor is not specified since optional.

The syntax of the various constructs introduced in CArtAgO is shown in Figure 2; meta-variables that have no definition (like *var*, *mstate*, or any identifier) correspond to constructs whose structure is not prescribed by CArtAgO architectural specification—their details depend on the actual integration of CArtAgO with existing languages to code agents (but also artifacts).

MAS and workspaces A multiagent system is a composition of agents ($\overline{g}$) with an environment of workspaces ($\overline{s}$). An agent can join and work in multiple workspaces at a time: when joining a workspace, an *agent body* is created as interface between the mind part – which depends on the specific agent model/architecture adopted – and the environment, and that contains sensors and effectors to enable interaction with artifacts belonging to that workspace. The same agent can work simultaneously in multiple workspaces, having a different agent body for each workspace. A workspace is hence formed by an identifier (σ), a set of artifacts ($\overline{a}$) and of agent bodies ($\overline{b}$). It is assumed that agent bodies and artifacts belong to a single workspace—the case where workspaces are not disjoined is not handled in this formalisation for the sake of simplicity.

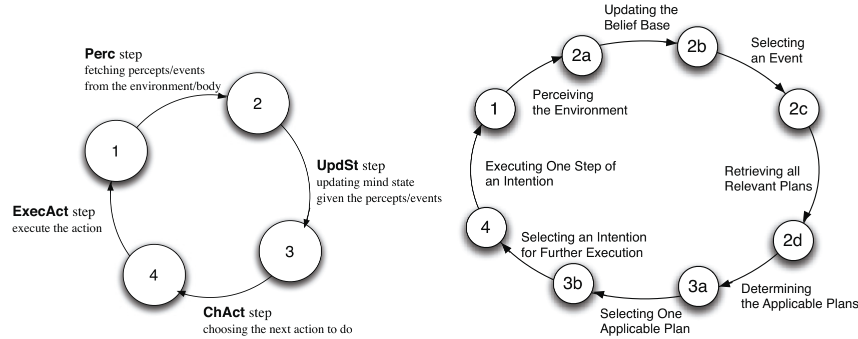


Fig. 3. *(left)* The stages of the agent cycle in the abstract model adopted in the formal model. *(right)* *Jason* (AgentSpeak) reasoning cycle adapted from [1], abstracting from stages related to message handling. The states are numbered so as to remark the relationships between the abstract model used in this paper and this model.

Agents An agent has an identifier (γ), and its state provides a label denoting the next step of the agent cycle to execute (*step*), a mind state (*mstate*), a percept state (*perstate*), a program (*prog*), and finally the set of workspaces it joined ($\overline{\sigma}$).

Analogously to agent reasoning cycle in *Jason* [1] or in approaches defining a common semantic basis for BDI languages [4], we structure agent behaviour in cyclic sequences of steps (stages). We consider only four steps, abstracting from those that are necessary to model aspects specifically related to the agent (cognitive) architecture: in Perc step, percepts and observable events are retrieved from the environment, updating the percept state of the agent; in UpdSt step, the mind state is updated according to the percept state of the agent; in ChAct step, an action to be executed is chosen; and finally in ExecAct step the chosen action is executed. The four stages are an abstraction over the steps found in concrete agent reasoning cycles. as an example, Fig. 3 on the right shows *Jason* (AgentSpeak) agent reasoning cycle, with states numbered so as to remark the relationships between the abstract model used in this paper.

Mind state is not directly modelled here for its details depends on the specific agent model/architecture. Typically, it would include a knowledge (beliefs) base, a goal base, a plan library, intentions stacks, and so on.

The percept state keeps track of the observable properties and events perceived by the agent, gathered in the Perc stage and fetched in the ChAct stage of each cycle. Percept state includes: *(i)* updates of observed properties ($\overline{prop}$), each specifying artifact identifier (α), property name (*pname*), and property value (*pvalue*); *(ii)* observable events generated by the artifacts that the agent is focussing, called artifact events ($\overline{artevt}$), each specifying artifact identifier (α), event value (*evtvalue*), and event type (*evttype*); and *(iii)* events related to agent actions executed in the past, called action events ($\overline{actevt}$), each specifying the executed action (*action*), a succeeded/failed label for the result (*res*), and an outcome information (*feedback*).

The agent program is abstracted in terms of a couple of functions: one that computes the next mind state from previous mind state and perceptions (*nextf*), and one that produces

the next action to execute (*actf*), analogously to the abstract model of state-full (intelligent) agents reported in [17] (chapter 2).

Actions This formalisation of CArtAgO handles a number of actions to be executed by agents, which are briefly described as follows: $\texttt{joinWsp}(\sigma)$ is used to join workspace σ, $\texttt{quitWsp}(\sigma)$ for leaving a workspace, and $\texttt{makeWsp}(\sigma)$ to create and join a new workspace; $\texttt{use}(\alpha, oname, opars, sid_{\perp}, timeout)$ triggers on artifact α an operation with name *oname*, fathering events in sensor $sid_{\perp}$ and with timeout *timeout*); $\texttt{sense}(sid, filter, timeout)$ gets from sensor *sid* an event that matches *filter* function, and with timeout *timeout*); $\texttt{focus}(\alpha, sid_perp)$ is used to start observing an artifact α gathering events in sensor *sid_perp*; analogously $\texttt{stopFocus}(\alpha)$ stops focussing; and finally, $\texttt{observeProp}(\alpha, pname)$ is issued to observe property *pname* in artifact α. It's worth remarking that when focussing an artifact, an agent automatically (and continuously) perceives as percepts from the environment the value of artifact observable properties and every observable events generated by artifact operation execution.

Other than standard success/failure semantics here actions use and sense can be *suspended* because of the condition of the dynamic context, and the result of action execution can be perceived by agent later in time: in the case of use when the artifact is working or the specified usage interface control is not currently enabled, in the case of sense when no events are currently found on the specified sensor.

To support environments with a dynamic and open structure, CArtAgO provides also actions for dynamically instantiating, disposing and looking up artifacts. Such actions are not part of the core model, since they are modelled as functionalities provided by special kind of artifacts, the *factory* and the *registry*, which are included by default in each workspace.

For sake of space in the following we do not handle timeouts that appear in use and sense actions: this feature can be modelled quite straightforwardly by introducing a local clock for each agent and timeouts as action failure events.

Artifacts and artifact templates An artifact has an identifier (α), a template identifier (τ), a logical name used for looking up (*aname*), a usage interface ($\overline{uic}$), observable properties ($\overline{prop}$), and finally an inner state (*instate*). The usage interface includes a set of controls, each providing the name of the operation it can trigger (*oname*) and a guard function over observable properties that enables the control (*guard*). The inner state has a label *wstate* denoting the execution mode (either idle or working), state variables ($\overline{var}$) and finally a set of pending operations ($\overline{o}$). Each operation has an identifier (ω), a name (as described by its usage interface control), a state label (*ostate*), and a context of local variables (*octx*).

An artifact is created out of a template; accordingly, in this model we assume the existing of a library of templates $\bar{t}_{\mathsf{lib}}$ to which any template t belongs. A template has an identifier (τ), a set of operation definitions (*opdef*), a definition of observable properties and variables (*prop* and *var*), and finally an initialization function and a manual. The definition of an operation basically provides a finite state automaton model, defining a set of state labels (*ostate*) including start and completed, a transition function (*onext*), and a guard function for transitions (*oguard*)—its details are described in next section.

Agent bodies An agent body is described by an identifier (β), the identifier (γ) of the agent it belongs to, sensors ($\overline{sns}$), pending artifact events for which no sensor was specified ($\overline{artevt}$), pending action events ($\overline{actevt}$), suspended actions ($\overline{action}$), and finally information on triggered operations ($\overline{otrig}$) and observed artifacts ($\overline{aobs}$). Sensors include a sensor identifier (sid) and a list of pending (artifact) events ($\overline{artevt}$); triggered operations specify an operation identifier (ω), artifact identifier (α), and the identifier for the optional sensor gathering events ($sid_{\perp}$); finally observed artifacts specify an artifact identifier (α) and the identifier for the optional sensor gathering events.

2.2 Dynamics

The dynamic aspects of CArtAgO are introduced through an operational semantics in Plotkin's style [11]. Namely, we provide a rule-based deduction system for formulas of the kind $mas \rightarrow mas'$, which describes a transition system for the state of multiagent systems.

We introduce some syntactic notation to make such rules more concise and readable. Meta-variable "$_$" is used that ranges over the set of all terms, and each use of this meta-variable is unique—as in Prolog anonymous variables, two occurrences of "$_$" in a rule are to be considered as different meta-variables, i.e., their content can be different. We abuse the notation "$\in$" to check whether the identifier of an abstraction has a corresponding declaration into a set of declarations, e.g., $\alpha \in \overline{a}$ would mean that $\overline{a}$ includes an artifact whose identifier is α. A conditional operator $?[cond]x$ is used to mean singleton x if condition $cond$ is true, empty set $\emptyset$ otherwise. As an example, $?[cond]x;?[\neg cond]y$ means x if $cond$ is true, and means y otherwise.

Some syntactic notation is also used for easily updating terms. Syntax $x \triangleright y$ means creating a clone of x with some changes as specified in y, and where a "$_$" in y means "no change". In this notation, x, y and $z = x \triangleright y$ are terms with the same structure: if y is "$_$" then $z = x$, otherwise $z = y$; then, if x (and y) is a compound term, function $\triangleright$ propagates recursively on arguments. For instance, $\langle a,b,c\rangle \triangleright \langle a,b',_\rangle$ would be $\langle a,b',c\rangle$, and $x \triangleright \langle _,_,a,_,_\rangle$ means a term equals to x in all arguments but third one, which should become a.

Finally, syntax $\overline{x}[\![x \triangleright\!\!\triangleright y]\!]$ is for multiple udpate in a set: it defines a set built from $\overline{x}$ by substituting all occurrences of elements that match x with a corresponding term $x \triangleright y$, for instance $a;a;b[\![a \triangleright\!\!\triangleright c]\!] = c;c;b$. Further conditions over x and y can be expressed into an optional "where" clause, for instance, term "$a(1);a(2);a(3)[\![a(X) \triangleright\!\!\triangleright b(X)]\!]$ where $\{X < 2\}$" is equal to $b(1);a(2);a(3)$.

Agent cyclic behaviour Figure 4 reports three rules handling key aspects of agent cyclic behaviour, namely perception, update and selection stages—actions execution is handled in the execution stage as described in the following sections.

First rule defines transitions labelled as *PERC*, handling the perception stage in which an agent g gathers all pertinent events from the environment. The bottom part states that a MAS moves from $\langle g;\overline{g},\overline{s}\rangle$ to $\langle g';\overline{g},\overline{s}'\rangle$, in that both the agent state g moves to new state g', and the workspace $\overline{s}$ move to $\overline{s}'$ due to changes in agent bodies—the other agents, denoted as $\overline{g}$, remain unchanged. In the top part, side conditions describe in which case g can change state, and what are the resulting g' and $\overline{s}'$. Let γ be the agent identifier, Perc its step, and $\overline{\sigma}$ the workspaces it joined – other agent fields are useless here –, g' is obtained from g by moving

$$\frac{\begin{array}{c} g = \langle \gamma, \mathsf{Perc}, _, _, _, \overline{\sigma} \rangle \qquad g' = g \triangleright \langle _, \mathsf{UpdSt}, _, \langle \overline{prop}, \overline{artevt}, \overline{actevt} \rangle, _, _ \rangle \\ \overline{prop} = \{ prop \in \overline{prop}' \,|\, \sigma \in \overline{\sigma}, \langle \sigma, \overline{a}, \overline{b} \rangle \in \overline{s}, \langle _, \gamma, _, _, _, _, _, \langle \alpha, _ \rangle; _ \rangle \in \overline{b}, \langle \alpha, _, _, _, \overline{prop}', _ \rangle \in \overline{a} \} \\ \overline{artevt} = \{ artevt \in \overline{artevt}' \,|\, \sigma \in \overline{\sigma}, \langle \sigma, \overline{a}, \overline{b} \rangle \in \overline{s}, \langle _, \gamma, _, \overline{artevt}', _, _, _, _ \rangle \in \overline{b} \} \\ \overline{actevt} = \{ actevt \in \overline{actevt}' \,|\, \sigma \in \overline{\sigma}, \langle \sigma, \overline{a}, \overline{b} \rangle \in \overline{s}, \langle _, \gamma, _, _, \overline{actevt}', _, _, _ \rangle \in \overline{b} \} \\ \overline{s}' = \overline{s}[\![\langle _, _, \overline{b} \rangle \rhd\!\rhd \langle _, _, \overline{b}' \rangle]\!] \text{ where } \{ \overline{b}' = \overline{b}[\![\langle _, \gamma, _, _, _, _, _, _ \rangle \rhd\!\rhd \langle _, _, _, \emptyset, \emptyset, _, _, _ \rangle]\!] \} \end{array}}{\langle g; \overline{g}, \overline{s} \rangle \xrightarrow{PERC} \langle g'; \overline{g}, \overline{s}' \rangle}$$

$$\frac{g = \langle \gamma, \mathsf{UpdSt}, mstate, perstate, \langle nextf, _ \rangle, _ \rangle \qquad g' = g \triangleright \langle _, \mathsf{ChAct}, nextf(mstate, perstate), \bot, _, _ \rangle}{\langle g; \overline{g}, \overline{s} \rangle \xrightarrow{UPDST} \langle g'; \overline{g}, \overline{s} \rangle}$$

$$\frac{g = \langle \gamma, \mathsf{ChAct}, mstate, _, \langle _, actf \rangle, _ \rangle \qquad g' = g \triangleright \langle _, \mathsf{ExecAct}, actf(mstate), _, _, _ \rangle}{\langle g; \overline{g}, \overline{s} \rangle \xrightarrow{CHOOSEACT} \langle g'; \overline{g}, \overline{s} \rangle}$$

Fig. 4. Agent cyclic behaviour: perception rule, update rule, and selection rule

the step to UpdSt and by setting the percept state to $\langle \overline{prop}, \overline{artevt}, \overline{actevt} \rangle$, where the three arguments are defined as follows. Changes to observable properties are obtained by joining all $\overline{prop}'$ from artifacts focussed by the agent: these are obtained as those artifacts whose identifier α occurs in the field "observed artifacts" of an agent body of γ—such agent bodies are searched into the workspaces joined by the agent, namely $\overline{\sigma}$. Both artifact and action events are obtained instead by gathering all pending events collected in the agent bodies: for the first case, for instance, we join all $\overline{artevt}'$ of the agent bodies in the environment that specify γ as agent identifier—and similarly for $\overline{actevt}$. Finally, the state of workspaces $\overline{s}$ is moved to $\overline{s}'$, by considering all bodies of agent γ and discharging their queues $\overline{artevt}$ and $\overline{actevt}$ of pending events.

The second rule labels transition as *UPDST*, as it changes an agent state due to the processing of perceptions. The step label moves from UpdSt to ChAct, and the mind state is recomputed thanks to function *nextf* of the agent program, which takes the current mind state *mstate* and the percept state *perstate* as computed by previous rule, and produces the new mind state—also, the percept state is discharged, and moved to $\bot$. No other changes to the MAS are applied.

The third rule labels transition as *CHOOSEACT*, since it computes the selection of an action to execute at next stage: this is simply achieved by changing mind state by program function *actf*—namely, changing mind state as a scheduler would do. Also, the step label moves from UpdSt to ChAct.

The last stage of the agent cycle is execution of action, moving the step label back to Perc, which comes in different rules depending on the selected action, as shown in the following.

Actions for workspace management We now describe how the actions reported in Figure 2 are executed in CArtAgO, and start with actions for managing workspace as show by rules in Figure 5.

First rule handles execution of action joinWsp(σ), by which an agent joins a workspace σ. Let g be an agent in the MAS that is in step ExecAct, with joinWsp(σ) being provided by the mind state as selected action—utility function selected_action is used to this end that extracts the selected action from agent mind. Let γ be the agent identifier and $\overline{\sigma}$ the workspaces it already joined. If σ is not already joined, the action succeeds, the agent step is moved back to Perc and σ is added to $\overline{\sigma}$; moreover, a new agent body is created for γ, with a new identifier β, and with empty sets for all fields (pending events, suspended actions, and so on). Such an

$$\frac{g = \langle \gamma, \mathsf{ExecAct}, mstate, _, _, \overline{\sigma} \rangle \qquad \mathsf{selected_action}(mstate) = \mathsf{joinWsp}(\sigma) \\ \sigma \notin \overline{\sigma} \qquad g' = g \triangleright \langle _, \mathsf{Perc}, _, _, _, \sigma\overline{\sigma} \rangle \qquad \mathsf{fresh}(\beta) \qquad \mathsf{b} = \langle \beta, \gamma, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle}{\langle g; \overline{g}, \langle \sigma, \overline{a}, \overline{b} \rangle \overline{s} \rangle \xrightarrow{EXECACT} \langle g'; \overline{g}, \langle \sigma, \overline{a}, b; \overline{b} \rangle \overline{s} \rangle}$$

$$\frac{g = \langle \gamma, \mathsf{ExecAct}, mstate, _, _, \sigma\overline{\sigma} \rangle \qquad \mathsf{selected_action}(mstate) = \mathsf{quitWsp}(\sigma) \\ g' = g \triangleright \langle _, \mathsf{Perc}, _, _, _, \overline{\sigma} \rangle \qquad \mathsf{b} = \langle _, \gamma, _, _, _, _, _, _ \rangle}{\langle g; \overline{g}, \langle \sigma, \overline{a}, b; \overline{b} \rangle \overline{s} \rangle \xrightarrow{EXECACT} \langle g'; \overline{g}, \overline{s} \rangle}$$

$$\frac{g = \langle \gamma, \mathsf{ExecAct}, mstate, _, _, \overline{\sigma} \rangle \qquad \mathsf{selected_action}(mstate) = \mathsf{makeWsp}(\sigma) \qquad \mathsf{fresh}(\sigma) \\ g' = g \triangleright \langle _, \mathsf{Perc}, _, _, _, \sigma\overline{\sigma} \rangle \qquad \mathsf{fresh}(\beta) \qquad \mathsf{s} = \langle \sigma, \mathsf{a}_{\mathsf{registry}}; \mathsf{a}_{\mathsf{factory}}; \mathsf{a}_{\mathsf{console}}, \langle \beta, \gamma, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle \rangle}{\langle g; \overline{g}, \overline{s} \rangle \xrightarrow{EXECACT} \langle g'; \overline{g}, s; \overline{s} \rangle}$$

Fig. 5. Workspace management actions: join, quit and make workspace

agent body is added to the workspace σ as shown in the bottom part of the rule. If σ is already present in $\overline{\sigma}$, the action fails and an event is properly generated—which is not formalised for the sake of simplicity and space: success and failure would be handled as shown for action use as shown in the next rules.

The second rule similarly handles exit from a workspace. In this case, σ is dropped from the workspaces joined by the agent, and moreover, the body of agent γ that occurs in σ is dropped from the environment.

Analogously, third rule describes the creation of a new workspace σ, which the agent joins. The new workspace s features three artifacts, namely a registry, a factory and a console[3] – which are supposed to be artifacts created out of given templates –, and a new body for agent γ, with a new identifier β and empty sets for all the others fields.

Use action Figure 6 shows four rules that handle use actions. First rule describes the semantics of action $\texttt{use}(\alpha, oname, opars, sid_\perp, timeout)$ executed by agent γ in the case of success, namely, the artifact α that is target of the action: *(i)* is in idle state (as reported in the *instate* field), and *(ii)* has a control for operation *oname* whose function *guard* (applied to observable properties $\overline{prop}$) is true. Let b be the body of agent γ into the workspace that hosts α, its sensors, action events, and triggered operations are updated as follows. First, if the action specifies an existing sensor, or does not specify a sensor (i.e. events directly reach the agent body), then sensors $\overline{sns}$ remain unchanged; otherwise a new sensor $\langle sid_\perp, \emptyset \rangle$ is created with empty queue. Second, a new action event is added to $\overline{actevt}$, specifying that the use action succeeded and yielding feedback ω, namely a newly generated operation identifier. Third, a new entry is added to triggered operations $\overline{otrig}$, specifying operation ω, artifact α, and sensor $sid_\perp$. Finally, the artifact state is changed to a', by moving its execution state to working and by adding an operation execution construct for ω.

Second rule deals with the case where operation *oname* does not exist in the usage interface. In this case, simply, a new action event is added to the agent body as in previous case; however, here the event reports a failure and an invalid operation identifier.

In third case, the action gets suspended in the agent body instead, which happens if the interface control exists but either the artifact is working or the guard is not positively evaluated. In this case, simply, the action is added to the "suspended actions" field of the agent body, and the agent gets back to a new cycle by step Perc.

[3] A console is a built-in artifact providing functionalities to write messages on standard output

$$\frac{\begin{array}{c} g = \langle \gamma, \mathsf{ExecAct}, \mathit{mstate}, _, _, \sigma;\overline{\sigma}\rangle \qquad \mathit{action} = \mathsf{selected_action}(\mathit{mstate}) = \mathtt{use}(\alpha, \mathit{oname}, \mathit{opars}, \mathit{sid}_\perp, _) \\ g' = g \triangleright \langle _, \mathsf{Perc}, _, _, _, _\rangle \qquad a = \langle \alpha, _, _, \langle [\mathit{guard}]\mathit{oname}\rangle; _, \overline{\mathit{prop}}, \langle \mathsf{idle}, _, \overline{o}\rangle\rangle \qquad \mathit{guard}(\overline{\mathit{prop}}) = \mathsf{true} \\ b = \langle _, \gamma, \overline{\mathit{sns}}, _, \overline{\mathit{actevt}}, _, \overline{\mathit{otrig}}, _\rangle \qquad b' = b \triangleright \langle _, _, \overline{\mathit{sns}}', _, \overline{\mathit{actevt}}', _, \overline{\mathit{otrig}}', _\rangle \qquad \mathsf{fresh}(\omega) \\ \overline{\mathit{sns}}' = \overline{\mathit{sns}}; ?(\mathit{sid}_\perp \neq \perp \text{ and } \langle \mathit{sid}_\perp, _\rangle \notin \overline{\mathit{sns}})\langle \mathit{sid}_\perp, \emptyset\rangle \qquad \overline{\mathit{actevt}}' = \overline{\mathit{actevt}}; \langle \mathit{action}, \mathsf{succeeded}, \omega\rangle \\ \overline{\mathit{otrig}}' = \overline{\mathit{otrig}}; \langle \omega, \alpha, \mathit{sid}_\perp\rangle \qquad a' = a \triangleright \langle _, _, _, _, _, \langle \mathsf{working}, _, \langle \omega, \mathit{oname}, \mathsf{start}, \mathit{opars}\rangle; \overline{o}\rangle\rangle \end{array}}{\langle g;\overline{g}, \langle \sigma, a;\overline{a}, b;\overline{b}\rangle \overline{s}\rangle \xrightarrow{\mathit{EXECACT}} \langle g';\overline{g}, \langle \sigma, a';\overline{a}, b';\overline{b}\rangle \overline{s}\rangle}$$

$$\frac{\begin{array}{c} g = \langle \gamma, \mathsf{ExecAct}, \mathit{mstate}, _, _, \sigma;\overline{\sigma}\rangle \qquad \mathit{action} = \mathsf{selected_action}(\mathit{mstate}) = \mathtt{use}(\alpha, \mathit{oname}, \mathit{opars}, \mathit{sid}_\perp, _) \\ g' = g \triangleright \langle _, \mathsf{Perc}, _, _, _, _\rangle \qquad a = \langle \alpha, _, _, \overline{\mathit{uic}}, _, _\rangle \qquad \langle [_]\mathit{oname}\rangle \notin \overline{\mathit{uic}} \\ b = \langle _, \gamma, _, _, \overline{\mathit{actevt}}, _, _, _\rangle \qquad b' = b \triangleright \langle _, _, _, _, \overline{\mathit{actevt}}', _, _, _\rangle \qquad \overline{\mathit{actevt}}' = \overline{\mathit{actevt}}; \langle \mathit{action}, \mathsf{failed}, \omega_\mathit{invalid}\rangle \end{array}}{\langle g;\overline{g}, \langle \sigma, a;\overline{a}, b;\overline{b}\rangle; \overline{s}\rangle \xrightarrow{\mathit{EXECACT}} \langle g';\overline{g}, \langle \sigma, a;\overline{a}, b';\overline{b}\rangle; \overline{s}\rangle\rangle}$$

$$\frac{\begin{array}{c} g = \langle \gamma, \mathsf{ExecAct}, \mathit{mstate}, _, _, \sigma;\overline{\sigma}\rangle \qquad \mathit{action} = \mathsf{selected_action}(\mathit{mstate}) = \mathtt{use}(\alpha, \mathit{oname}, \mathit{opars}, \mathit{sid}_\perp, _) \\ a = \langle \alpha, _, _, \langle [\mathit{guard}]\mathit{oname}\rangle \overline{\mathit{uic}}, \overline{\mathit{prop}}, \mathit{instate}\rangle \qquad (\mathit{instate} = \langle \mathsf{working}, _, _\rangle \text{ or } \mathit{guard}(\overline{\mathit{prop}}) = \mathsf{false}) \\ g' = g \triangleright \langle _, \mathsf{Perc}, _, _, _, \sigma;\overline{\sigma}\rangle \qquad b = \langle _, \gamma, _, _, _, \overline{\mathit{action}}, _, _\rangle \qquad b' = b \triangleright \langle _, _, _, _, _, \mathit{action};\overline{\mathit{action}}, _, _\rangle \end{array}}{\langle g;\overline{g}, \langle \sigma, a;\overline{a}, b;\overline{b}\rangle; \overline{s}\rangle \xrightarrow{\mathit{EXECACT}} \langle g';\overline{g}, \langle \sigma, a';\overline{a}, b';\overline{b}\rangle; \overline{s}\rangle}$$

$$\frac{\begin{array}{c} b = \langle _, _, \overline{\mathit{sns}}, _, \overline{\mathit{actevt}}, \mathit{action};\overline{\mathit{action}}, \overline{\mathit{otrig}}, _\rangle \qquad \mathit{action} = \mathtt{use}(\alpha, \mathit{oname}, \mathit{opars}, \mathit{sid}_\perp, _) \\ a = \langle \alpha, _, _, \langle [\mathit{guard}]\mathit{oname}\rangle; _, \overline{\mathit{prop}}, \langle \mathsf{idle}, _, \overline{o}\rangle\rangle \qquad \mathit{guard}(\overline{\mathit{prop}}) = \mathsf{true} \\ b' = b \triangleright \langle _, _, \overline{\mathit{sns}}', _, \overline{\mathit{actevt}}', \overline{\mathit{action}}, \overline{\mathit{otrig}}', _\rangle \qquad \mathsf{fresh}(\omega) \\ \overline{\mathit{sns}}' = \overline{\mathit{sns}}; ?(\mathit{sid}_\perp \neq \perp \text{ and } \langle \mathit{sid}_\perp, _\rangle \notin \overline{\mathit{sns}})\langle \mathit{sid}_\perp, \emptyset\rangle \qquad \overline{\mathit{actevt}}' = \overline{\mathit{actevt}}; \langle \mathit{action}, \mathsf{succeeded}, \omega\rangle \\ \overline{\mathit{otrig}}' = \overline{\mathit{otrig}}; \langle \omega, \alpha, \mathit{sid}_\perp\rangle \qquad a' = a \triangleright \langle _, _, _, _, _, \langle \mathsf{working}, _, \langle \omega, \mathit{oname}, \mathsf{start}, \mathit{opars}\rangle \overline{o}\rangle\rangle \end{array}}{\langle \overline{g}, \langle \sigma, a;\overline{a}, b;\overline{b}\rangle; \overline{s}\rangle \xrightarrow{\mathit{UNBLOCK}} \langle \overline{g}, \langle \sigma, a';\overline{a}, b';\overline{b}\rangle; \overline{s}\rangle}$$

Fig. 6. Rules for the use action: success, failure, suspension and unblock

$$\frac{\begin{array}{c} g = \langle \gamma, \mathsf{ExecAct}, \mathit{mstate}, _, _, _\rangle \qquad \mathit{action} = \mathsf{selected_action}(\mathit{mstate}) = \mathtt{sense}(\mathit{sid}, \mathit{filter}, _) \\ b = \langle _, \gamma, \langle \mathit{sid}, \overline{\mathit{artevt}}\rangle; \overline{\mathit{sns}}, _, \overline{\mathit{actevt}}, _, _, _\rangle \qquad \mathit{artevt} = \mathit{filter}(\overline{\mathit{artevt}}) \neq \perp \qquad g' = g \triangleright \langle _, \mathsf{Perc}, _, _, _, _\rangle \\ b' = b \triangleright \langle _, _, \langle _, \overline{\mathit{artevt}} \setminus \mathit{artevt}\rangle; \overline{\mathit{sns}}, _, \overline{\mathit{actevt}}', _, _, _\rangle \qquad \overline{\mathit{actevt}}' = \overline{\mathit{actevt}}; \langle \mathit{action}, \mathsf{succeeded}, \mathit{artevt}\rangle \end{array}}{\langle g;\overline{g}, \langle \sigma, \overline{a}, b;\overline{b}\rangle; \overline{s}\rangle \xrightarrow{\mathit{EXECACT}} \langle g';\overline{g}, \langle \sigma, \overline{a}, b';\overline{b}\rangle; \overline{s}\rangle}$$

$$\frac{\begin{array}{c} g = \langle \gamma, \mathsf{ExecAct}, \mathit{mstate}, _, _, _\rangle \qquad \mathit{action} = \mathsf{selected_action}(\mathit{mstate}) = \mathtt{sense}(\mathit{sid}, \mathit{filter}, _) \\ b = \langle _, \gamma, \langle \mathit{sid}, \overline{\mathit{artevt}}\rangle \overline{\mathit{sns}}, _, _, \overline{\mathit{action}}, _, _\rangle \qquad \mathit{filter}(\overline{\mathit{artevt}}) = \perp \qquad g' = g \triangleright \langle _, \mathsf{Perc}, _, _, _, _\rangle \\ b' = b \triangleright \langle _, \gamma, _, _, _, \mathit{action};\overline{\mathit{action}}, _, _\rangle \end{array}}{\langle g;\overline{g}, \langle \sigma, \overline{a}, b;\overline{b}\rangle; \overline{s}\rangle \xrightarrow{\mathit{EXECACT}} \langle g';\overline{g}, \langle \sigma, \overline{a}, b';\overline{b}\rangle; \overline{s}\rangle}$$

$$\frac{\begin{array}{c} b = \langle _, _, \langle \mathit{sid}, \overline{\mathit{artevt}}\rangle \overline{\mathit{sns}}, _, \overline{\mathit{actevt}}, \mathit{action};\overline{\mathit{action}}, _, _\rangle \qquad \mathit{action} = \mathtt{sense}(\mathit{sid}, \mathit{filter}, _) \\ \mathit{artevt} = \mathit{filter}(\overline{\mathit{artevt}}) \neq \perp \qquad b' = b \triangleright \langle _, _, \langle \mathit{sid}, \overline{\mathit{artevt}} \setminus \mathit{artevt}\rangle \overline{\mathit{sns}}, _, \overline{\mathit{actevt}}', \overline{\mathit{action}}, _, _\rangle \\ \overline{\mathit{actevt}}' = \overline{\mathit{actevt}}; \langle \mathit{action}, \mathsf{succeeded}, \mathit{artevt}\rangle \end{array}}{\langle \overline{g}, \langle \sigma, \overline{a}, b;\overline{b}\rangle; \overline{s}\rangle \xrightarrow{\mathit{UNBLOCK}} \langle \overline{g}, \langle \sigma, \overline{a}, b';\overline{b}\rangle; \overline{s}\rangle}$$

Fig. 7. Rules for sense action: success, suspension, and unblock

The latter rule is executed when one such suspended action can finally be executed, namely, when the guard is true and the artifact is idle. Accordingly, the artifact, as well as sensors, action events, and triggered operations in the agent body are updated as in the case of success (first rule).

Sense action As a use action is successfully executed, the artifact will eventually produce observable events that are collected by the agent bodies: from then, the agent can perceive such events through a sense action, whose semantics is shown in the three rules of Figure 7.

First rule handles successful execution of action $\mathtt{sense}(\mathit{sid}, \mathit{filter}, \mathit{timeout})$. Let g be the executing agent and *action* the executing action as usual, and let b be the body of g having a sensor with identifier *sid* as specified in *action*. If the queue of events attached to the sensor has an event *artevt* that matches function *filter*, then the action successfully executes, in which

$$\frac{\begin{array}{c} g = \langle \gamma, \mathsf{ExecAct}, mstate, _, _, _\rangle \qquad action = \mathsf{selected_action}(mstate) = \mathtt{focus}(\alpha, sid_\perp) \qquad \alpha \in \overline{a} \\ g' = g \triangleright \langle _, \mathsf{Perc}, _, _, _, _\rangle \qquad b = \langle _, \gamma, \overline{sns}, _, \overline{actevt}, _, _, \overline{aobs}\rangle \qquad b' = b \triangleright \langle _, _, \overline{sns}', _, \overline{actevt}', _, _, \overline{aobs}'\rangle \\ \overline{sns}' = \overline{sns}; ?(sid_\perp \neq \perp \text{ and } \langle sid_\perp, _\rangle \notin \overline{sns})\langle sid_\perp, \emptyset\rangle \qquad \overline{actevt}' = \overline{actevt}; \langle action, \mathsf{succeeded}, \perp\rangle \\ \overline{aobs}' = \overline{aobs}; ?(\langle \alpha, _\rangle \notin \overline{aobs})\langle \alpha, sid_\perp\rangle \end{array}}{\langle g; \overline{g}, \langle \sigma, \overline{a}, b; \overline{b}\rangle; \overline{s}\rangle \xrightarrow{EXECACT} \langle g'; \overline{g}, \langle \sigma, \overline{a}, b'; \overline{b}\rangle; \overline{s}\rangle}$$

$$\frac{\begin{array}{c} g = \langle \gamma, \mathsf{ExecAct}, mstate, _, _, _\rangle \qquad action = \mathsf{selected_action}(mstate) = \mathtt{stopFocus}(\alpha) \qquad \alpha \in \overline{\mathtt{a}} \\ g' = g \triangleright \langle _, \mathsf{Perc}, _, _, _, _\rangle \qquad b = \langle _, \gamma, _, \overline{actevs}, _, _, _, \overline{aobs}\rangle \qquad b' = b \triangleright \langle _, _, _, \overline{actevs}', _, _, _, \overline{aobs}'\rangle \\ \overline{actevs}' = \overline{actevs}; \langle action, \mathsf{succeeded}, \perp\rangle \qquad \overline{aobs}' = \overline{aobs} \setminus \langle \alpha, _\rangle \end{array}}{\langle g; \overline{g}, \langle \sigma, \overline{a}, b; \overline{b}\rangle; \overline{s}\rangle \xrightarrow{EXECACT} \langle g'; \overline{g}, \langle \sigma, \overline{a}, b'; \overline{b}\rangle; \overline{s}\rangle}$$

$$\frac{\begin{array}{c} g = \langle \gamma, \mathsf{ExecAct}, mstate, _, _, _\rangle \qquad action = \mathsf{selected_action}(mstate) = \mathtt{observeProp}(\alpha, pname) \\ g' = g \triangleright \langle _, \mathsf{Perc}, _, _, _, _\rangle \qquad a = \langle \alpha, _, _, _, \overline{prop}, _\rangle \qquad b = \langle _, \gamma, _, \overline{actevt}, _, _, _, _\rangle \\ b' = b \triangleright \langle _, _, _, \overline{actevt}; \overline{actevt}'; \overline{actevt}'', _, _, _, _\rangle \\ \overline{actevt}' = ?(\langle _, pname \mapsto pvalue\rangle \in \overline{prop})\langle action, \mathsf{succeeded}, pvalue\rangle \\ \overline{actevt}'' = ?(\langle _, pname \mapsto _\rangle \notin \overline{prop})\langle action, \mathsf{failed}, \mathsf{invalid_prop}\rangle \end{array}}{\langle g; \overline{g}, \langle \sigma, a; \overline{a}, b; \overline{b}\rangle; \overline{s}\rangle \xrightarrow{EXECACT} \langle g'; \overline{g}, \langle \sigma, a; \overline{a}, b'; \overline{b}\rangle; \overline{s}\rangle}$$

Fig. 8. Focussing, stop focussing, and observe actions

case that event is dropped from the queue, and a new action event is generated in the body, stating the successful execution of *action* and yielding event *artevt* as feeback.

The second rule similarly handles the case where no event matches the filter, in which case simply the action is added to the "suspended actions" field of the agent body.

The third rule instead checks whether there is an agent body with a suspended sense whose filter now actually finds a pending event. If this is the case, the body is updated as in the case of a successful sense (first rule).

Focussing and observation actions Figure 8 reports rules providing operational semantics to actions for focussing and observation.

First rule handles action $\mathtt{focus}(\alpha, sid_\perp)$, by which an agent γ can focus on artifact α, and receive all related events on sensor $sid_\perp$. Let σ be the workspace where α is hosted, and b the body of agent γ as occurring in σ, then the fields $\overline{sns}$, $\overline{actevt}$, and $\overline{aobs}$ of b are changed as follows. In $\overline{sns}$, a new sensor is generated if needed as in rule for action use; in $\overline{actevt}$, a new event stating that *action* succeeded is added; and finally in $\overline{aobs}$, a new entry is added (if not already there) to state that α is now focussed on sensor $sid_\perp$.

Second rule handles the dual action $\mathtt{stopFocus}(\alpha)$. In this case, other than adding to $\overline{actevt}$ in b a new event stating that *action* succeeded, entry $\langle \alpha, _\rangle$ is dropped from $\overline{aobs}$.

Third rule handles action $\mathtt{observeProp}(\alpha, pname)$, by which the agent wants to get information about the current value of property *pname* of artifact α. Let $\overline{prop}$ be the properties of α, $\overline{actevt}$ of b is added with a new event that states success of action if $\overline{prop}$ include *pname* – in which case the corresponding value is returned as feedback – or states failure otherwise.

These rules conclude the description of last step of agent cycle, completing semantics of action execution.

Artifact management Figure 9 describes rules providing the operational semantics of artifacts. First rule makes a working artifact executing an internal computational step, which changes internal state possibly generating observable events which are collected in agent bodies, second rule then selects a new execution step.

$$
\frac{\begin{array}{c}
a = \langle \alpha, \tau, _, _, \overline{prop}, instate \rangle \qquad instate = \langle \mathsf{working}, \overline{var}, \bar{\mathsf{o}}; \langle \omega, oname, ostate, ocxt \rangle \rangle \\
\langle \tau, \overline{opdef}; \langle oname, _, onext, oguard \rangle, _, _, _, _ \rangle \in \bar{t}_{\mathsf{lib}} \qquad oguard(ostate, octx, \overline{prop}, \overline{var}) = \mathsf{true} \\
onext(ostate, octx, \overline{prop}, \overline{var}) = \langle ostate', octx', \overline{prop}', \overline{var}', \overline{e} \rangle \qquad a' = a \triangleright \langle _, _, _, _, \overline{prop}', instate' \rangle \\
instate' = \langle \mathsf{idle}, \overline{var}', \bar{\mathsf{o}}; ?(ostate' \neq \mathsf{completed}) \langle \omega, oname, ostate', ocxt' \rangle \rangle \\
b = \langle _, _, \overline{sns}, \overline{artevt}, _, _, \overline{otrig}; \langle \omega, \alpha, sid_\perp \rangle, _ \rangle \qquad b' = b \triangleright \langle _, _, \overline{sns}', \overline{artevt}', _, _, \overline{otrig}', _ \rangle \\
\overline{e}_{toadd} = \overline{e}; ?(ostate' = \mathsf{completed}) \langle \alpha, \omega : \mathsf{op_exec_completed} \rangle \\
\overline{sns}' = ?(sid_\perp = \perp) \overline{sns} ?(sid_\perp \neq \perp) \overline{sns} [\![\langle sid_\perp, \overline{e}_0 \rangle \rhd\!\rhd \langle sid_\perp, \overline{e}_0; \overline{e}_{toadd} \rangle]\!] \\
\overline{artevt}' = \overline{artevt}; ?(sid_\perp = \perp) \overline{e}_{toadd} \qquad \overline{otrig}' = \overline{otrig}; ?(ostate' \neq \mathsf{completed}) \langle \omega, \alpha, sid_\perp \rangle \\
\overline{b}' = \overline{b} [\![\langle _, _, \overline{sns}_0, \overline{artevt}_0, _, _, _, \langle \alpha, _ \rangle; _ \rangle \rhd\!\rhd \langle _, _, \overline{sns}_1, \overline{artevt}_1, _, _, _, _ \rangle]\!] \text{ where} \{ \\
\quad \overline{sns}_1 = ?(sid_\perp = \perp) \overline{sns}_0; ?(sid_\perp \neq \perp) \overline{sns}_0 [\![\langle sid_\perp, \overline{e}_0 \rangle \rhd\!\rhd \langle sid_\perp, \overline{e}_0; \overline{e} \rangle]\!] \\
\quad \overline{artevt}_1 = \overline{artevt}_0; ?(sid_\perp \neq \perp) \overline{e}' \\
\}
\end{array}}{\langle \overline{g}, \langle \sigma, a; \overline{a}, b; \overline{b} \rangle; \overline{s} \rangle \xrightarrow{EXECSTEP} \langle \overline{g}, \langle \sigma, a'; \overline{a}, b'; \overline{b}' \rangle; \overline{s} \rangle}
$$

$$
\frac{\begin{array}{c}
a = \langle \alpha, \tau, _, _, \overline{prop}, instate \rangle \qquad instate = \langle \mathsf{idle}, \overline{var}, \bar{\mathsf{o}}; \langle \omega, oname, ostate, ocxt \rangle \rangle \\
\langle \tau, \overline{opdef}; \langle oname, _, _, oguard \rangle, _, _, _, _ \rangle \in \bar{t}_{\mathsf{lib}} \qquad oguard(ostate, octx, \overline{prop}, \overline{var}) = \mathsf{true} \\
a' = a \triangleright \langle _, _, _, _, _, instate' \rangle \qquad instate' = instate \triangleright \langle \mathsf{working}, _, \bar{\mathsf{o}}; \langle \omega, oname, ostate, ocxt \rangle
\end{array}}{\langle \overline{g}, \langle \sigma, a; \overline{a}, \overline{b} \rangle; \overline{s} \rangle \xrightarrow{SELSTEP} \langle \overline{g}, \langle \sigma, a'; \overline{a}, \overline{b} \rangle; \overline{s} \rangle}
$$

Fig. 9. Artifact behaviour: step execution and selection

We start considering first rule—which is rather involved for it encapsulates most of artifact behaviour. Let a be an artifact in working state, ω a pending operation with name *oname*, *guard* its guard function and *onext* the update function of ω (modelling its computation) as specified in the artifact template τ. Operation ω can be selected if the guard, applied to *(i)* operation state, *(ii)* operation context, *(iii)* artifact properties and *(iv)* artifact inner variables, yields true. In this case, function *onext* is applied to such 4 elements, yielding new values for them as well as a set of artifact events to be fired that we denote as $\overline{e}$. The artifact is then updated: its state moves to idle, variables and properties are updated, and finally the operation ω is either dropped (if its new state is completed) or its state and context are updated.

At this point, the body b of the agent that executed the use action should be updated in the fields for sensors, artifact events, and triggered operations. We first let $\overline{e}_{toadd}$ be the set of artifact events to be added, which is the set of events $\overline{e}$ generated by *onext*, plus an event of operation completion if the reached state is completed. Now, sensors are updated only if the sensor was specified ($sid_\perp \neq \perp$), in which case we simply add events $\overline{e}_{toadd}$ to the sensor $sid_\perp$—note that the multiple update operator applies to one element only. If the sensor was not specified instead, $\overline{e}_{toadd}$ is added to the field $\overline{artevt}$ of b. Then, in the set of triggered operations we keep ω only if the operation is not completed.

Other than updating the body of the agent that executed the use action, however, we should also notify all bodies of those agents that focussed on the artifact. So, we take the set of bodies $\overline{b}$ in current workspace, and (by a multiple update operator) change sensors and artifact events of those bodies that focus on α. Such changes are much the same of those we did for b, though here we do not add the operation completion event, hence we use $\overline{e}$ in place of $\overline{e}_{toadd}$.

Finally, the second rule applies if the artifact state is idle and if at least one operation ω is pending such that the *oguard* function is true. If this is the case we simply move the artifact state to working, so that an execution step will be selected and executed as described by previous rule.

3 Discussion

In this section we focus on some features of the computational model that are put in evidence in the formalisation and that we consider relevant for MAS programming in general.

3.1 Artifacts vs. Objects

The formalisation makes it possible to clarify the core aspects that characterise the artifact abstraction and agent-arttifact interaction model, and then to remark the differences with respect to existing abstractions adopted in programming languages: in particular here we focus on the *object* abstraction, as defined in Object-Oriented Programming.

Similarly to objects, artifacts encapsulate a state and provides an interface for interacting with them, as non-autonomous entities. Such an interface, however, in the case of artifacts is *not* composed by methods that are invoked with control-coupling between the caller and the callee, like in the case of objects, possibly returning a value. On the contrary, as described by rules in 6 and 9, there is no control coupling between the agent action (use) triggering the execution of an operation and operation execution. Also, there is no returning value: output information generated by operation execution are made perceivable to agents as observable events, and multiple events can be generated (vs. a single return value), distributed in time.

Then, the basic model of objects does not include any *observable* state: the only way to interact with an object is by calling methods though its interface—the use of public fields is considered bad programming, violating encapsulation and information hiding principles. On the contrary, artifacts support as a fundamental feature the possibility to expose a set of observable properties, which can change over the time and whose value can be perceived by agents *without acting upon the usage interface*—this point will be deepened in Subsection 3.3.

Finally, the basic OO model does not include mechanisms for dealing with concurrency: on the contrary, the computational model of artifacts has been conceived with concurrency in mind – next subsection deepens this point – being multi-agent systems concurrent systems.

3.2 Concurrency and Control

Concurrency is a main issue to consider with for any model of environment in multi-agent systems, both from a theoretical and practical point of view. On the one side, full concurrency must be allow for actions of agents acting on independent parts of an environment; on the other side, interactions must be effectively managed (avoided, in the case of interferences) for actions of agents working on parts of an environment that have some kind of dependency.

The computational model presented in this paper allows for the concurrent access and use of artifacts by multiple agents, without interferences. Multiple operations can be triggered – and executed – on the same artifact concurrently; however, only one operation step at a time is executed—following rules in Figure 9, a single operation step is first selected (artifact in idle state) and then executed (artifact in working state). This makes it possible to avoid a-priori interferences and race conditions due to simultaneous access of the artifact state. Consequently, MAS programmers don't have to think to these low-level problems when designing agents/artifacts, since they are solved natively by the computational model.

Then, distinct artifacts of the same work environment can be exploited by agents completely in parallel—formally, this is apparent from the fact that the rules for action execution on distinct artifacts (e.g. use action) do not have dependencies, as well as those concerning operation step selection and execution of artifacts themselves. So, under this perspective artifacts can be exploited to encapsulate parallel tasks/processes of the environment, fully controllable by agents through properly designed usage interfaces.

3.3 Properties of the Observation Mechanism

The observation mechanism, as main part of the interaction model, has some distinct features compared to message-based communication based on ACL, making it particularly effective (and efficient) to support forms of knowledge sharing and coordination strategies based knowledge sharing.

First, the observation by an agent of an observable property of an artifact by means of an observeProp action has no effect on the state and processes inside the artifact, i.e. artifact's state is not changed and no actions on the artifact side are executed as a result of the action. This is evident by looking at the third rule in Figure 8: the action is executed in spite of the state of the artifact a and the transition does not change the state of the artifact. This implies that – generally speaking – reading the value of an observable property by one or multiple agents on an artifact is more efficient compared to the exchange of messages adopted in solutions purely based on agents and direct communication. This because the processing of a received message by an agent typically requires – at least in cognitive agents – the update of the belief base (with the new message), the generation of a new event, the selection of a prope r plan to manage the event, the access – by the plan execution – of the current value of the belief storing the knowledge and finally the execution of a communicative action to respond. So, by referring to the formal model, multiple steps of the agent (reasoning) cycle are needed, each one carrying a specific cost. In the case of artifacts, no computational steps are actually required.

The interested reader can find in Section A in the appendix a simple example which shows this result in practice. The example is about sharing a certain piece of knowledge $info$ – that can change dynamically – among a set of agents. A classic solution in agent programming is introducing a *mediator* agent, which makes the information available to interested agents by means of a request-response communication protocol. A solution based on artifacts account for introducing an artifact *KB* with a $info$ observable property, a usage interface with an operation to update it, and interested agents reading the property by means of observeProp. We implemented and tested both the solutions using the *Jason* platform (i.e. AgentSpeak language), in one case with only agents (former solution) and in one case exploiting the integration with CArtAgO. The test shows that the solution based on the artifact is more efficient compared the solution purely based on agents.

Then, analogous benefits can be devised when considering continuous observation of artifacts, realised by the *focus* mechanism (first rule in Figure 8). As an example, consider an extension of previous example where there is not only knowledge sharing but also coordination strategies such as – for instance – publish/subscribe protocols, where agents must be notified each time a certain knowledge changes. Without using artifacts, the mediator agent must be extended to receive/manage the subscriptions and send the notifications as soon as the knowledge changes. With artifacts, *KB* artifact keeps the same and the responsibility of the

continuos observation is decentralised upon observing agents exploiting focus action. This is supported automatically by the CArtAgO machinery at two different levels: at the event level, by dispatching observable events generated by a step execution to the bodies of the agents focussing the artifact (first rule in Figure 9); at the percept level, by refreshing at each agent cycle the percept state of the agent, including the current state of the observable properties of (only) the artifact(s) observed by the agent (first rule in Figure 4). Note that this provides an easy and high-level way to *select* – on the agent side – which part of the environment to observe, i.e. the subset of percepts the agent is interested in, without necessarily specifying filters.

3.4 Properties of the Use Mechanism

As a dual aspect of observation, use mechanism provides basic features which make it effective to synchronise agent actions with environmental resources and finally other agents' activities. As described by the first rule in Figure 6, the execution of use action involves a *synchronisation* between the user agent and the used artifact: if the action succeeds, the agent can conclude that the operation started. The action is suspended if the usage interface control is not enabled, until it eventually becomes enabled (or a timeout occurs). This basic semantics can be exploited to realise effective higher-level synchronisation policy, without the need of introducing complex ACL-based communication protocols.

As a simple yet effective and general example, consider a *producer-consumer* scenario, where *N* agents produce repeatedly some kind of information items that must be processed by any of *M* consumer agents. Producers and consumers agents can have different speeds in doing their tasks: then, some mechanism/strategy must be introduced to coordinate the overall work, and it must effective both for the performance (time) and the resource (e.g. memory, network) consumed. The problem can be solved by introducing a kind of *bounded inventory* (bounded buffer) to uncouple the interaction of producers and consumers and, at the same time, to synchronise their activities, providing a locus of design (the size of the inventory) for tuning the performance of the system. Thanks to the basic synchronisation support provided by the computational model, the implementation of the bounded inventory using CArtAgO, as well as the implementation of producers and consumers agents exploiting it is straightforward (the interested reader can find the source codes of the artifacts and of the agents implemented in *Jason* in Section B in the appendix). In particular, the usage interface of the artifact would simply include two usage interface controls to respectively insert (`put`) and consume (`get`) items, the former with a guard specifying that the control is enabled if (when) the buffer is not full and the latter when the buffer is not empty.

4 Conclusion and Future Works

In this paper we introduced a formalisation for artifact-based environments, whose aim is to both clarify the semantics of artifact computational and programming model, and to foster the integration of CArtAgO with existing agent programming languages, in particular with those with a well-defined formal model and semantics.

In future work, we aim at using the formal model to *(a)* provide a more complete and formal account of the basic properties that the model has in general – informally discussed

in Section 3; *(b)* set up an executable specification in frameworks such as the MAUDE term-rewriting language, so as to investigate the verification of correctness properties of MAS behaviour based on artifact-based environments, by exploiting analysis tools like LTL model-checking.

References

1. R. Bordini, J. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. John Wiley & Sons, Ltd, 2007.
2. S. Bromuri and K. Stathis. Situating Cognitive Agents in GOLEM. In D. Weyns, S. Brueckner, and Y. Demazeau, editors, *Engineering Environment-Mediated Multiagent Systems (EEMMAS'07)*. LNCS Springer, Oct 2007. to appear.
3. M. Dastani, D. Hobo, and J.-J. Meyer. Practical extensions in agent programming languages. In *In Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'07)*. ACM Press, 2007.
4. L. A. Dennis, B. Farwer, H. R. Bordini, M. Fisher, and M. Wooldridge. A common semantic basis for bdi languages. In *Programming Multi-Agent Systems*, number 4908 in Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2007.
5. S. Eker, J. Meseguer, and A. Sridharanarayanan. The maude ltl model checker. *Electr. Notes Theor. Comput. Sci.*, 71, 2002.
6. J. Ferber and J.-P. Müller. Influences and reaction: a model of situated multi-agent systems. In *Proc. of the 2nd International Conference on Multi-Agent Systems (ICMAS'96)*. AAAI, 1996.
7. K. V. Hindriks, F. S. De Boer, W. Van Der Hoek, and J.-J. Ch. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.
8. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.
9. A. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni. The KGP model of agency. In *Proc. of the 16th European Conference of Artificial Intelligence*, pages 33–37, 2004.
10. A. Omicini, A. Ricci, and M. Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17 (3), Dec. 2008.
11. G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.
12. A. S. Rao. AgentSpeak(l): BDI agents speak out in a logical computable language. In *MAAMAW '96: Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : agents breaking away*, pages 42–55, Secaucus, NJ, USA, 1996. Springer-Verlag New York.
13. A. Ricci, M. Piunti, M. Viroli, and A. Omicini. Environment programming in CArtAgO. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah-Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications, Vol. 2*. Springer Verlag, 2009. To appear.
14. A. Ricci, M. Viroli, and A. Omicini. The A&A programming model & technology for developing agent environments in MAS. In M. Dastani, A. El Fallah Seghrouchni, A. Ricci, and M. Winikoff, editors, *Post-proceedings of the 5th International Workshop "Programming Multi-Agent Systems" (PROMAS 2007)*, volume 4908 of *LNAI*, pages 91–109. Springer, 2007.
15. D. Weyns and T. Holvoet. Formal model for situated multiagent systems. *Fundamenta Informaticae*, 63(2–3):125–158, 2004.
16. D. Weyns and H. V. D. Parunak, editors. *Journal of Autonomous Agents and Multi-Agent Systems. Special Issue: Environment for Multi-Agent Systems*, volume 14(1). Springer Netherlands, 2007.
17. M. Wooldridge. *An Introduction to Multi-Agent Systems*. John Wiley & Sons, Ltd, 2002.

A Observation Test

The observation test is about sharing a certain piece of knowledge (that changes dynamically) among a set of agents. The solutions are implemented on the *Jason* platform integrated with CArtAgO. The source code as well as all the necessary information and technology to execute the tests can be downloaded from the CArtAgO web site.

The first solution (Table 1) adopts only agents: *N* instances of *requester* agents – who plays the roles of observers – continuously ask an `info` information to a *mediator* agent, who is responsible to keep and manage the shared knowledge (which is updated by the mediator 100 times, an update each 100 ms). Requester agents go on asking the information until the information read achieves the value 100 (which happens – given the behaviour of the mediator agent – after a fixed amount of time, about 10 seconds). To evaluate the performance, we counted the total number of reads successfully executed by the requesters: the higher the number is, more efficient is the solution. We did six tests, with *N* ranging from 1 to 25 (1, 5, 10, 15, 20, 25), i.e. with an increasing number of requester agents concurrently accessing the shared knowledge. Actually the solution comes in two variants, the one – shown in Table 1 – in which in the mediator agent a plan is specifically written to handle the request, and a second one – not reported – in which we exploit the built-in capabilities of *Jason* agents to automatically send the answer if a belief exactly corresponding to the content of `askOne` performative is found. In the performance results reported in Fig. 10 the two variants are labelled `(a)` and `(b)`.

The second solution (reported in Table 2) uses a `KB` artifact instead of a mediator, with an `info` observable property containing the shared value and a usage interface with the `update` operation to update the value. An *updater* agent is used to create the artifact and periodically update its value, analogously to the mediator agent. *N* instances of *observer* agents – with *N* again ranging from 1 to 25 in 6 steps – continuously observe the `info` property of the artifact.

The tests have been executed using *Jason* version 1.2, CArtAgO version 1.3.5, on top of a Java platform 1.5.0, running on a Apple MacBook Pro with an Intel Core 2 duo 2.33 Ghz and 2 GB of RAM. Fig. 10 shows the results we got, comparing the performance of the solution with the artifact (the line with rhombuses), the first variant with only agents and an explicit plan to handle requests (the line with square) and the second variant with only agents

```
// mediator agent
kbinfo(0).
!do_test.

+!do_test : true <- !update.
+!update : kbinfo(N) & N < 100
  <- .wait(100);
     -+kbinfo(N+1); !update.
+!update : myinfo(100)
  <- .my_name(Me); .kill_agent(Me).
+!kqml_received(S, askOne, info, R) :
  kbinfo(Info)
   <- .send(S,tell,Info, R).
```

```
// requestor agent
ncount(0).
current_value(0).
!do_test.

+!do_test : true <- !request.
+!request : current_value(X) & X < 100
  <- .send(mediator,askOne,info,Reply);
     -ncount(N); +ncount(N+1);
    -+current_value(Reply); !request.
+!request : current_value(100)
  <- -ncount(N); .print(N);
     .my_name(Me); .kill_agent(Me).
```

Table 1. Source code of the *mediator* and *requester* agent(s). The requester agents repeatedly ask the mediator for the value of the shared information and, as soon as the value achieves 100, they print it on standard output and terminate. The values printed on standard output – reported then in Fig. 10 – represent the number of reads successfully executed.

```
// KB artifact
public class KB extends Artifact {
  void init(){ defineObsProperty("info",0); }
  @OPERATION void update(int v){ updateObsProperty("info", v); }
}
```

```
// updater agent
ncount(0).
!do_test.

+!do_test : true
  <- cartago.makeArtifact("kb","KB",KB);
       !update.
+!update : ncount(N) & N < 100
  <- cartago.use(kb,update(N));
      .wait(100); -+ncount(N+1); !update.
+!update : ncount(100)
  <- cartago.use(kb,update(100));
      .my_name(Me); .kill_agent(Me).
```

```
// observer agent
ncount(0).
current_value(0).
!do_test.

+!do_test : true
  <- cartago.lookupArtifact("kb",KB); !observe.
+!observe : current_value(X) & X < 100
  <- cartago.observeProperty(kb,info(Value));
      -ncount(N); +ncount(N+1); -+current_value(Value);
      !observe.
+!observe : current_value(100)
  <- -ncount(N); .print(N);
      .my_name(Me); .kill_agent(Me).
```

Table 2. Source code of the KB artifact and of the *updater* and *observer* agent(s). The observer agents repeatedly observe the observable property in the KB artifact and, as soon as the value achieves 100, they print it on standard output and terminate.The values printed on standard output – reported then in Fig. 10 – represent the number of reads successfully executed. .

and built-in *Jason* plan to manage incoming messages (the line with triangles). In all cases the solution based on the artifact outperforms the solution purely based on agents—in both variants: the performance improvement is particularly evident when the number of observers (requesters) agents is greater than 20.

B Producers-Consumers Test

Fig. 11 shows the implementation of a *bounded inventory* artifact in CArtAgO, exploiting guards in usage interface controls (put and get) to synchronise agent use of the inventory. A sketch of the *producer* and *consumer* agents implemented in *Jason* follows:

```
!produce.
+!produce <-
  ?nextItemToProduce(Item);
  cartago.use(store,put(Item));
  !produce.

+?nextItemToProduce(Item) : true
  <- ..
```

```
!consume.
+!consume: true <-
  cartago.use(store,get,s0);
  cartago.sense(s0,new_item(Item));
  !consumeItem(Item);
  !consume.

+!consumeItem(Item) : true
  <- ...
```

Thanks to the basic synchronisation support provided by the use action, the agents need not to use further mechanisms or protocols to coordinate their actions in using the inventory (so as to avoid to insert elements if the inventory is full and retrieve elements if the inventory is empty).

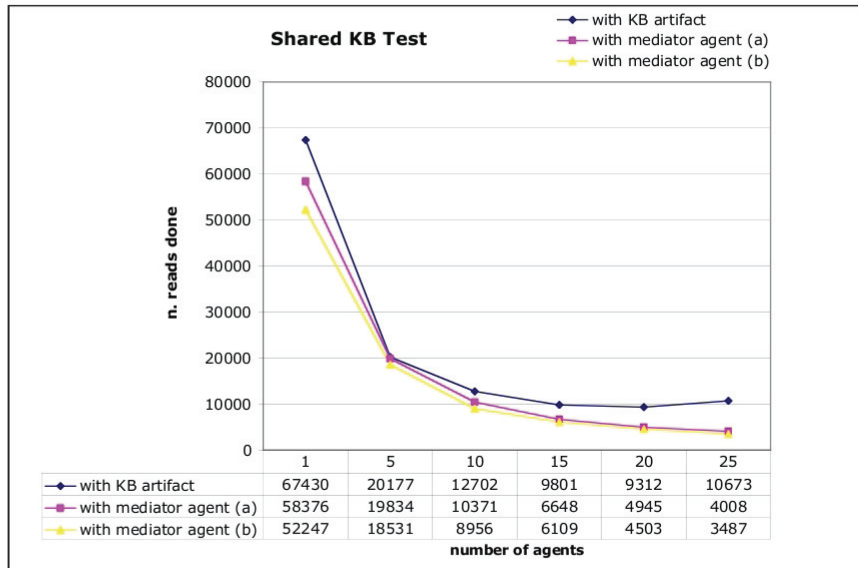


Fig. 10. Results of the tests: the values on the Y axis represents the number of reads that observer (requester) agents were able to complete.

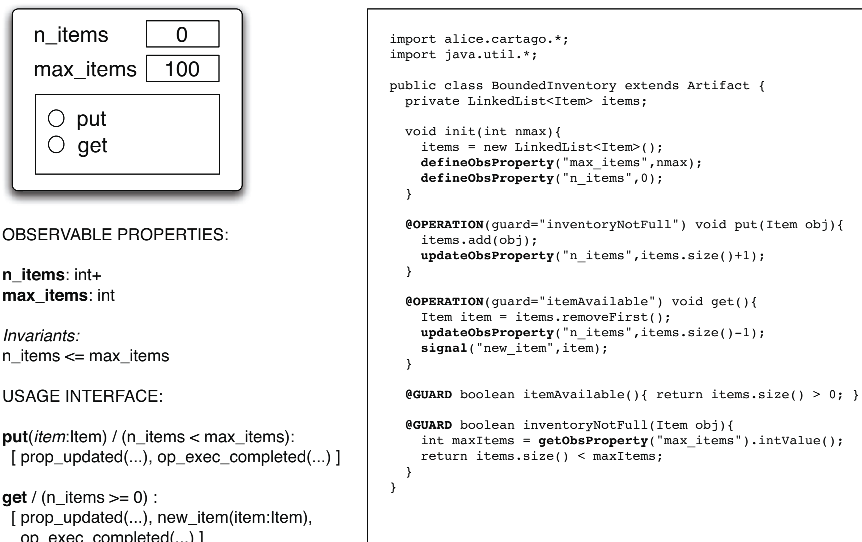


Fig. 11. A simple bounded-inventory artifact, exploiting guards in usage interface controls to synchronise agent use of the inventory.

Infrastructure for forensic analysis of multi-agent based simulations

Emilio Serrano, Juan A. Botia and Jose M. Cadenas
emilioserra@um.es, juanbot@um.es, jcadenas@um.es

University of Murcia**, Murcia, Spain

Abstract. The Multi Agent Systems (MAS) theory has methodical approaches to analyze, understand and debug the social level of agents. This paper aims to explain that technologies used in the analysis of MAS can be used for Multi-agent based simulation (MABS). In particular, *forensic analysis* is proposed. It is explained the creation of an infrastructure for forensic analysis to assist the analysis of any model independently of its scope and framework of development . To achieve this genericity, the proposal is based in the use of Aspect Oriented Programming (AOP). In addition, it is given the key ideas to implement this infrastructure on the MABS platform MASON, giving a great power of analysis to this framework.

1 Introduction

Multi-agent based simulation, MABS, is used in more and more scientific domains [7]: sociology, biology, physics, chemistry, ecology, economy, etc. where it is progressively replacing previous simulation techniques. It is due to its ability to model very different "individuals", starting from simple entities to more complex one. Its versatility makes MABS one of the most favorite and interesting support for the simulation of complex systems [7].

Fishwick [8] defines computer simulation as the discipline of designing a model of a system, executing the model on a computer, and analyzing the execution output. We think that the testing of the social behavior of the agents groups, in the analysis task, is one of the most interesting thing about these simulations. This is to analyze the macro-social perspective. But, generally, researchers are more interested in model design [7]. The model execution and the execution analysis are viewed as a less scientific work. We consider necessary the development of methodical proposals for the task of analyzing the MABS because with a bad model execution or a bad execution analysis, even the best of the models cannot provide reliable and useful results.

** This research work is supported by the Spanish Ministry of Education and Science in the scope of the Research Project TIN-2005-08501-C03-02 and by the Project "Análisis, Estudio y Desarrollo de Sistemas Inteligentes y Servicios Telemáticos" through the Fundación Séneca within the Program "Generación del Conocimiento Científico de Excelencia".

As already mentioned, the scope of MABS is increasing. In addition to the increasing of application domains of MABS, it have also proliferated lots of MABS frameworks for their development. The web of the Open Agent Based Modeling Consortium[1] lists 17 of these frameworks, including: MASON [12], Repast [3] and NetLogo [1]. Because of the analysis is an unavoidable task, all platforms provide approaches to this task. In the section of related work are some of these proposals. The first and obvious defect is that these proposals do not offer compatibility between platforms. It is obvious that the more developers use a technology to debug systems, the more benefit is going to get the developers and the technology itself (which receives feedback from the first ones). Therefore, genericity in technologies for analysis and debugging is an important factor to consider. The next defect which make us look for new proposals is that even in the early stages of analysis, data collection, is often required specific knowledge about how the model under review has been programmed. These technical skills in computer sciences are very different from the knowledge of the specific domain of the model which usually is required for the responsible for the analysis. The ideal is to have an infrastructure for the analysis of any model and abstracted of the specific programming.

The immediate question is whether it is possible to automate or at least assist the process of analyzing a MABS regardless of the specific application domain and the way in which it was programmed. It seems difficult to find properties in common, for example, inside a model about biology and one about economy. Herbert Simon explains how patterns often appear between different complex systems in his article *The Architecture of Complexity* [20]. For example, it explains how the complexity often takes the form of hierarchy in the sense that the complex systems are composed by subsystems which are composed of subsystems and so on. Automating, or at least assisting, the process of discovering these social structures in any model is a powerful tool. In the case of discovering hierarchies, for example, results permit to scale the analysis of a complex system in a "divide and conquer" approach. In the field of Multi Agent Systems have been proposed methods to discover hierarchies in systems independently of the specific application domain [19].

The field of *Multi Agent Systems* (MAS), a well-established branch of AI, is complementary in several aspects to MABS [4]. The MAS theory has methodical approaches to analyze, understand and debug the social level of agents. This paper aims to explain that technologies used in the analysis of MAS can be used for MABS. In particular, *forensic analysis* is proposed. Forensic analysis is the process of understanding, re-creating, and analyzing arbitrary events that have occurred previously [15]. This technology has already been used successfully to debug MAS in a social level [18]. This paper explains how to create an infrastructure for forensic analysis to assist the analysis of any model independently of its scope and with the flexibility of choosing the MABS framework. This infrastructure also supports the use of representations that help to analyze and understand the MABS in the same way as with the MAS.

[1] OpenABM Consortium website: http://www.openabm.org/

Next section treat related works and introduces the main approaches to analyze and debug systems in the fields of MABS and MAS. Then, section 3 explains how to create an infrastructure for forensic analysis of MABS providing it with flexibility over the model and platform. Section 4 details the key ideas in the implementation of forensic analysis for MASON. Finally, conclusions and future work are given.

2 Related works

To a greater or lesser extent, forensic analysis is present in any MABS platforms because the analysis of the simulations is an essential task. MASON [12] is criticized for providing limited facilities for such purposes [17]. It allows developers to save properties of an agent or model in a text file, but you cannot do something as simple as saving the value of two properties in the same file. On the other hand, MASON allows developers to save a whole execution as checkpoints to relaunch simulations later. However, you cannot access to previous states in the stored simulation. On the opposite side is NetLogo [1], praised for its ability to record a great variety of events in simulations [17]. In particular, NetLogo permits a special execution which logs the simulation in a xml file. Details can be found in the Logging section of the *NetLogo 4.0.4 User Manual* [1]. Specifically, there are 8 loggers available for different types of events: *globals* (a global variable change), *greens* (elements of the interface chage), *code* (NetLogo code is comiled), *widgets* (widgets added/removed from the interface), *buttons* (...pressed or released), *speed-slider* (...changes), *turtles* (...die or are born), *links* (...die or are born). Surprisingly, little information is given about the agents ("turtles" on NetLogo), only when they are born or they die. It is also surprising that the most revealing element of the NetLogo interface, plots, are not stored as event using the logger *greens*. These shortcomings are compensated by allowing the export of plots or the simulated world to spreadsheets. Having different approaches to log events on the same platform is a little confusing. In addition, these approaches are often too rigid and hardly allow configuration. About Repast [3] is said that it is the most complete platform [17] and it is really the most powerful in the analysis task because it provides the best collecting of events. Although Repast delegates several external tools for analysis, which means all the tools have to be known, the data collecting is always performed by "Datasets". The datasets can be constructed as a series of values that are obtained from calls to methods of the agents (or formulas composed of these calls). The problem is that this proposal requires a deep knowledge about the specific programming of the specific model because the user has to distinguish between methods that make reference to relevant properties and methods that are merely ancillary. That is a trivial task in simple models, but in complex models may be impractical. As mentioned in the introduction, the problem of these approaches is that in the first place they have not been concerned about the compatibility between platforms, something very interesting to study the task of analysis itself and independently of the platform in which the models

are programmed. The other major problem, most pronounced in Repast, is that the collection of data often requires specific knowledge of how the model was programmed. This means that the researcher in charge of the analysis should know the details of the model programming.

Simulations typically generates huge amounts of data, the analysis of simulations is a very complex issue which deserves especial attention. In the MAS theory, we can find literature that deals with the analysis of the agents. In this way, Serrano et al. [18] details the creation of an infrastructure for the forensic analysis which is flexible about the used MAS platform and which do not require deep knowledge of the specific system programming. Forensic analysis is often the basis of most proposals about debugging in MAS and it is often accompanied by understandable data representations. There are works which analyze the behavior of the agents groups with *petri nets* [5], *AUML diagrams* [16], extensions of the *propositional dynamic logic* [14], *statecharts* [10], *dooley graphs* [13], etc. In general, the more possibilities of automation in analysis is having by the representations, the more complicated are these representations to be developed and understood by humans. These papers illustrate different infrastructures for forensic analysis in MAS which can be used in MABS. However, the objective of the analysis in these work is the understanding of the agents in their group level, not the social level which is interesting in MABS.

Beyond the group level, the field of MAS has researched the analysis of the social level of agents, which is intimately related to the analysis of MABS. Analyzing a MAS as a society refers to check if some properties are accomplished during the life time of the society [9]. The scope of such properties is the whole system, not individual or group components. The fundamental difference in the social level of MAS with their group level is that emergent properties can appear without being specified in the design. This is especially interesting for MABS [4]. It has been of interest, for example, to detect emergent behaviors in agents societies which were not included in the specification of a group of agents in the MAS design [6]. There are works that combine forensic analysis, graphs theory, and data mining to achieve simple representations of the agent society. In this way, Serrano et al. [19] have detailed the creation of graphs that reflect the collaborative cores of agents or similarity among members of the agent society. These representations are obtained independently of the specific system, the MAS platform and without requiring programming skills to be generated or understood. This type of representation can help to analyze the behavior of the system, understand it, debug it and even to identify emergent behaviors. The basis of these representations is the basic forensic analysis infrastructure which is presented in this paper to be used in MABS.

3 Debugging MABS from a MAS forensic analysis

3.1 Infrastructure for forensic analysis

The analysis of MAS in the agent level of is not very different from traditional software debugging. However, most approaches to debug a MAS in its group level

are based on the forensic analysis [15], simply because it is not as interesting the analysis of a specific execution as the overall analysis of a significant group of executions of the system. This is because the frequent presence of randomness makes irrelevant the result of a concrete execution in isolation. A fundamental reason for the use of forensic analysis appears in the analysis of the social level of agents: detecting behaviors that have not previously been defined, as emergent behaviors. Because of the possibility of these unpredictable behaviors which make impossible define preconditions and post conditions to validate the system, most conventional approaches for the analysis, testing and debugging are ineffective. The resulting database of forensic analysis supports technologies for discovering not explicit knowledge, such as exploratory data mining, which can assist the analysis of MABS to find no predefined behaviors.

To a lesser or greater extent forensic analysis is present in any MABS platforms. As seen in related work, the proposals are often aimed at a specific platform, a specific model and they usually requires knowledge about the programming of the model. Serrano et al. [18] details the creation of an infrastructure for forensic analysis in MAS that can be reused for MABS. This proposal provides flexibility over the specific developed system and development platform. Besides, users of the analyzer do not need to know the details of the system implementation. The key idea is to capture interesting elements of a simulation and store them in a relational database (RDB) to allow making consults and getting simplified representations of the stored data. These representations and consults assist the analysis process. Figure 1 shows the analyzer that records data in a RDB and a developer consulting that RDB and studying representations to analyze MABS. To bring this forensic analysis from the MAS to MABS, the immediate question is which elements are interesting to register in a MABS.

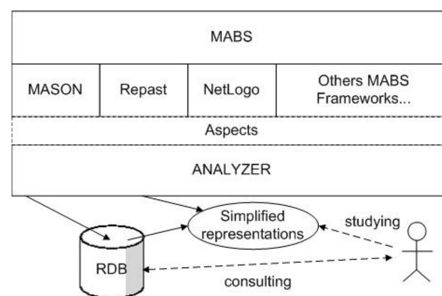


Fig. 1. Proposal of the paper

The related works section explains the approaches of some MABS platforms to collect the execution data. The closest thing to a common element to register for all MABS in these approaches was the creation of agents by the logger "turtles" in Netlogo. In fact, it is not even required in all MABS that the agents

are located in a simulated world after creating them. It is really difficult to find common elements in the broad domain of application of MABS. Therefore, platforms usually require the programmer of the models to collect relevant information. However, the developer can use whatever framework and develop whatever model, but he always is going to use the facilities that the framework offers. In this way, elements that always appear are displays, for example, *"inspectors"* in MASON or *"monitors"* in NetLogo. These displays show certain properties of the model in general or the agents in particular which the modeler or programmer considered relevant. Hence, regarding what elements are interesting to capture in a simulation for a forensic analysis, it can be said that all those elements which were chosen by the developers to be monitored in the simulation are clearly relevant. Any type of interesting event can also be stored, common to any platform and model (such as creating agents) or not. The ideal is to store these properties in an RDB, which is a powerful tool of querying and can assist analysis of MABS permitting to make consultations and supporting exploratory data mining. As seen above, this proposal allows each domain has its own taxonomy of events instead of imposing one. Therefore, it is the responsibility of the modeler to determine events of interest in the model and the programmer must simply use the facilities of the framework to show these events in the simulation.

3.2 Implementing the infrastructure

Regarding how to capture elements of the simulation, to capture an interesting event first must be located the point in the code of the platform or model in which the event occurs. Then, the event must be stored in this point. It could be thought that it is trivial programming calls to RDB or model into the platform to include the necessary code to store the events. This classic approach presents a major problem; this code would serve only for the specific model or the specific version of the platform. With the number of frameworks, and their continuous changes, this implementation would have maintenance very costly. In addition, the registration code would be dispersed throughout the model or platform code. Therefore, every change in the code of forensic analysis would be very expensive because it should be replicated in practically any platform or model. The solution in this proposal is the use of *Aspect oriented programming (AOP)* [11]. AOP can isolate the aspect of register a MABS execution in certain classes which are called *"aspects"*. Then, the aspects can be programmed or modified separately and in a modular manner. Aspects are programmed to capture interesting events in models implemented in some platform. If the platform is upgraded, it only has to be recompiled to include the aspects. If the strategy of forensic analysis changes, then the only point to change is the aspects (without changing the model or the platform code). The cost of maintenance is greatly reduced because the code of the forensic analysis is centralized in a few aspects. The only restriction is the requirement of the MABS platform source code for the compilation including the aspects. Furthermore, although it is not necessary to change the points of interest where events occur, the code of the platform should be understandable to locate these points and to program aspects. Figure 1 shows how AOP is the layer that

joins a MABS platform with the analyzer. Even the programming language of the platform is not a limitation. AspectJ, which is used for Java codes, is the most popular AOP language, but there are others like AspectC, AspectC++, AspectC#... for C, C++ y C# respectively. With a code isolated in a few areas and the freedom to choose the programming language, the goal of not being limited to a particular MABS platform is satisfactorily achieved. Moreover, as we have seen, the cost of maintaining the infrastructure for the forensic analysis is reduced using AOP and this cost might be, as in any software, the most expensive part of its life cycle.

As mentioned, an infrastructure for forensic analysis of MABS based in AOP is not limited by the MABS framework chosen either the developed model. However, one interesting question is whether the infrastructure for a framework can be reused for others. It is not difficult to find the equivalence of the elements and concepts of a platform in others frameworks, a famous comparison of MABS platforms [17] even published a table in that regard. Specifically, the *"Graphical display"* concept is an inspector in MASON, a monitor in NetLogo and a probe in Repast. Using this concept, it can be registered for the forensic analysis those elements which the developers of the modelers considered relevant to the analysis. The forensic analysis can also be expanded with other concepts, such as the "agent location" (field in MASON, world in NetLogo and space in Repast) to analyze the positions of the agents at each timestep, although that concept not necessarily is going to appear in all models. Besides, if the developers did not add displays of the agent locations, it would indicates the irrelevance of this concept in the analysis phase. With a clear equivalence of the fundamental concepts of MABS platforms and a code of forensic analysis isolated in aspects, reprogramming the aspects for other platforms is easy. In general and as noted above, the only restriction is that the MABS frameworks have to be open source software to use AOP. Figure 1 shows how the analyzer can be used on different platforms. With the flexibility provided by this proposal, we hope this work will be useful for the MABS community independently of the specific preferences for programming.

3.3 MABS framework considerations

Forensic analysis is suitable for any MABS platform, as stated in section 3.2, portability of forensic analysis between MABS platforms is an idea that always has been in our research. However, there are platforms in which the implementation is more or less useful and more or less simple. MASON [12], Repast [3] and NetLogo [1] were considered for the first implementations, although as noted in the introduction, only in the web of the Open Agent Based Modeling Consortium are listed 17 frameworks. There are several papers where MABS platforms are compared. Steven F. Railsback et al. [17] do not dare to recommend one platform over another, but they give conclusions on each of the frameworks. It is indicated that NetLogo is the most usable, Mason is the fastest platform and Repast is the most complete.

NetLogo cannot implement the proposal of this paper because it do not give the source code necessary for the inclusion of aspects, so it was quickly discarded until this code is available. There are also a feature of NetLogo and Repast which makes MASON a little more compatible with the forensic analysis, the faithful reproduction of the experiments. NetLogo and Repast do not provide methods to reproduce the order in which an action is executed on a list of agents. In a forensic analysis, after detecting anomalies at certain points in certain simulations, the researcher should be able to re-execute the simulations in these points to find out what could happen. However, this iterative approach is less effective if it is not ensured the same result in a repeated simulation than in a registered simulation. For this reason, we chose Mason for the first implementation of forensic analysis. Furthermore, the distribution of Repast software is confusing and has not clearly separated the core from the domain-specific applications [17]. This property complicates the location of the points which the aspects have to refer to. However, the solution is simply a deeper study of the Repast platform. Other comparisons are more determined, Matthew Berryman [2] concludes that the best platform is Repast. Repast, an open source framework with a large community of developers, is undoubtedly the ideal candidate to implement the infrastructure for forensic analysis after MASON.

4 SAM, social analyzer for MASON

As noted above, the analysis phase of MABS has been traditionally performed in an exploratory and intuitive manner [7]. One of the weaknesses of MASON, and MABS platforms in general, is that they offer few facilities to monitor and debug the simulated models [17]. MASON offers the possibility of fix inspectors in individual properties (of the model or of the agents) to be monitored / recorded / modified from the simulation. However, we miss many options as views and records of the artificial society as a whole. On the other hand, the programmer can make his own tests for a specific simulation, for example, a property of the model can be an array with all that interesting properties of the agents or the average of some properties. To improve this approach, which is poor and rigid, is created SAM, social analyzer for MASON. This section introduces the key ideas to implement SAM, an infrastructure for forensic analysis in MASON using AOP.

Before discussing the code of specific aspects of forensic analysis is necessary to explain what is an aspect. This information can be expanded in the on line documentation for AspectJ [2]. In very basic terms, the essential concepts to understand the AOP are join point, pointcut, advice and aspect. A method call *join point* encompasses the actions of an object receiving a method call. It includes all the actions that comprise a method call, starting after all arguments are evaluated up to and including return (either normally or by throwing an exception). *Pointcuts* pick out certain join points in the program flow, but they don't do anything apart. Then, to actually implement a behavior, advices must

[2] AspectJ website: http://www.eclipse.org/aspectj

be used. An *Advice* brings together a pointcut (to pick out join points) and a body of code (to run at each of those join points). The combination of the pointcut and the advice is termed an *aspect*. Now, some MASON concepts have to be explained to program useful aspects in this framework.

To program the aspects in MASON, the first step is to define the interesting join points for the forensic analysis. As stated, we are interested in what the developer of a simulation chose to be shown in the MASON simulation. This is the concept of *inspector* in MASON and its location in the MASON code is extremely clear. Any method that starts with "get" in the code of an agent (implements the *Steppable* interface) or a model (implements *SimState*) returns a property that is displayed when this agent or model respectively is being inspecting. Now, aspects to add behavior to MASON must be programmed.

The following *pointcuts* pick out the join points where an inspector is called to show a property.

```
pointcut modelInspector(SimState model):
  target(model) && call(public get*());
pointcut agentInspector(Steppable agent):
  target(agent) && call(public get*());
```

The "target" pointcut picks out each join point where the target object (the object on which a method is called or a field is accessed) is an instance of a particular type, SimState or Steppable in these cases. And `‘‘call(public get*()’’` picks out all call join points to public methods where the method name starts with the characters "get".

In the same way, we can define pointcuts for the time of creating agents in the model which, as seen in the previous section, is one of the few elements common to all MABS.

```
pointcut newModel: call(SimState+.new());
pointcut newAgent: call(Steppable+.new());
```

These pointcuts picks out all constructor call join points where an instance of any subtype of a SimState or Steppable, respectively, is constructed. Once pointcuts are defined, we need now to specify the functionality for each; this is what is called advices. Their structure would be like this

```
after (SimState model) returning (Object r):
  modelInspector(model){
      /*mySniffingCode...*/
  }
after (Agent agent) returning (Object r):
  agentInspector(agent){
     /*mySniffingCode...*/
  }
```

in which we define an advice just after calling a model inspector and just after calling a agent inspector, respectively.

In the body of code, calls to some RDB can be made to store information about the object returned by the inspector (defined in aspect as the variable ``r''), about the model itself (defined as the variable ``model'') or on the agent (defined as the variable ``agent''). Similarly, advices for the moments of creating models or agents can be defined with the following structure.

```
after(): newModel{
  /*mySniffingCode...*/
}
after(): newAgent{
  /*mySniffingCode...*/
}
```

in which we define an advice just after creating a new model and just after creating a new agent, respectively. Notice that, due to the use of powerful regular expressions allowed by the AOP, we can define pointcuts for any model or framework with a few lines of code. To illustrate this flexibility by way of example, we can define the following pointcuts

```
pointcut myMasonClasses():
  within(SimState) || within(Steppable);
pointcut myMasonConstructor():
  myMasonClasses() && execution(new(..));
pointcut myMasonMethod():
  myMasonClasses() && execution(* *(..));
```

which pick out each method and constructor which is called by the classes SimState and Steppable of MASON.

It can be seen as the code for forensic analysis is simple and flexible. Besides all this code is in a only class aspect, isolated from the model or platform code, facilitating its maintenance. Moreover, the dependence on the framework is minimal, only a few lines of code shown in this section. Once this basic analysis has be covered, it can be captured anything of interest for forensic analysis. In any case, having all properties inspected in an RDB gives a powerful capacity of analysis to MASON, allowing to make consultations and operations using SQL like averages, standard deviations, number of different values in a property, etc. Besides the RDB supports intelligent data analysis technologies and understandable representations which can be very useful to assists the analysis of MABS.

5 Conclusions and future work

The paper introduces an infrastructure to assist the analysis, understanding and debugging of a *Multi-agent based simulation* (MABS) extrapolated of *multi-agent*

systems (MAS). The infrastructure consists of forensic analysis by *aspect oriented programming* (AOP). The infrastructure is flexible about the used MABS framework, the simulated model and how the model is programmed.

The paper starts explaining how researchers in the field of MABS have neglected the analysis phase, despite the fact that the success of a research in this field depends on it. All MABS platforms have approaches to the analysis, but they depend on the platform, the specific model, and the specific programming of the model. However, approaches to the analysis task which resolves these problems has been given in the field of MAS. That is why this paper aims to extrapolate the use of these approaches to MABS, specifically, a forensic analysis based on AOP is proposed. The key idea is to capture interesting elements of a simulation using AOP and store them in a relational database (RDB) to allow consultations or to get simplified representations of the stored data. The proposal directly delegates the developers to discover what elements are interesting to register. Then, the developers must use the facilities of the MABS framework to show these elements. The use of AOP can isolate all the analysis code in a few classes called "aspects". In this way, the analysis code is flexible about changes in the chosen platform, the platform version, the model code, the analysis code, etc. With the adaptation to these situations, the maintenance cost is greatly reduced. The paper concludes giving the keys to the concrete implementation of the infrastructure for the MASON platform, providing MASON with a great power of analysis. The flexibility and genericity in the proposal is illustrated with only a few lines of AOP.

Regarding future work, the immediate one is to migrate the forensic analysis from MASON to Repast which has a large developer community. Another important work is to obtain, from the forensic analysis of MABS, representations to simplify the data and to provide an analysis of certain aspects of the agent society. Specifically, we want to use the graphs to reflect collaborative cores of agents and similarity among members of society [19], mentioned in related works. There is a great variety of representations in the field of MAS that can assist the analysis of the MABS. We also intend to investigate the automation of the analysis process (when it is possible). Once again, the field of MAS has many approaches to model interactions between agents and then, they can be automatically tested and validated. However, understandable representations are always necessary to help humans to discover unexpected and not modeled elements as emergent behaviors. In general, we are interested in integrating our work with any technology that helps to analyze, understand and debug the social level of agents in MABS.

References

1. Netlogo 4.0.4 user manual. web: http://ccl.northwestern.edu/netlogo/docs/.
2. Matthew Berryman. Review of software platforms for agent based models. *DSTO Defence Science and Technology Organisation*, 2008. Australian Government, Department of Defence.
3. N. Collier. Repast: An extensible framework for agent simulation. 2002.

4. Rosaria Conte, Nigel Gilbert, and ao Sichman Jaime Sim˙ Mas and social simulation: A suitable sommitment. In *Proceedings of the First International Workshop on Multi-Agent Systems and Agent-Based Simulation*, pages 1–9, London, UK, 1998. Springer-Verlag.
5. R. Scott Cost, Ye Chen, Timothy W. Finin, Yannis Labrou, and Yun Peng. Using colored petri nets for conversation modeling. In *Issues in Agent Communication*, pages 178–192, London, UK, 2000. Springer-Verlag.
6. Nuno David, Jaime Simão Sichman, and Helder Coelho. Towards an emergence-driven software process for agent-based simulation. In *3rd International Workshop on Multi-Agent Based Simulation (MABS)*, 2002.
7. Alexis Drogoul, Diane Vanbergue, and Thomas Meurisse. Multi-agent based simulation: Where are the agents? In Jaime S. Sichman, Francois Bousquet, and Paul Davidsson, editors, *Proceedings of the Third International Workshop on Multi-Agent-Based Simulation MABS 2002, Bologna, Italy*, LNAI 2581, pages 1–15. Springer Verlag, Berlin Heidelberg, July 2002.
8. Paul A. Fishwick. Computer simulation: growth through extension. *Trans. Soc. Comput. Simul. Int.*, 14(1):13–23, 1997.
9. Jorge J. Gómez, Juan A. Botia, Emilio Serrano, and Juan Pavón. Testing and debugging of mas interactions with ingenias. Agent oriented software engineering. Workshop at AAMAS'08, 2008. Estoril, Portugal.
10. David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, Inc., New York, NY, USA, 1998.
11. Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.
12. Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, and Keith Sullivan. Mason: A new multi-agent simulation toolkit. In *Proceedings of the 2004 Swarmfest Workshop*, 2004.
13. H. Van Dyke Parunak. Visualizing agent conversations: Using enhanced dooley graphs for agent design and analysis. *Proc. Second Int'l Conf. Multiagent Systems, AAAI Press, Menlo Park, Calif.*, pages 275–282, 1996.
14. Shamimabi Paurobally. Developing agent interaction protocols using graphical and logical methodologies. In *PROMAS, volume 3067 of LNCS*, pages 149–168. Springer, 2003.
15. Sean Philip Peisert. *A model of forensic analysis using goal-oriented logging*. PhD thesis, La Jolla, CA, USA, 2007. Adviser-Karin, Sidney.
16. David Poutakidis, Lin Padgham, and Michael Winikoff. Debugging multi-agent systems using design artifacts: the case of interaction protocols. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 960–967, New York, NY, USA, 2002. ACM.
17. Steven F. Railsback, Steven L. Lytinen, and Stephen K. Jackson. Agent-based simulation platforms: Review and development recommendations. *SIMULATION*, 82(9):609–623, September 2006.
18. Emilio Serrano and Juan A. Botia. Infrastructure for forensic analysis of multi-agent systems. volume Proceedings of the Programming Multi-Agent Systems Workshop at AAMAS'08, 2008. Estoril, Portugal.
19. Emilio Serrano, Jorge J. Gomez-Sanz, Juan A. Botia, and Juan Pavon. Intelligent data analysis applied to debug complex software systems. *Neurocomputing*. To appear.
20. Herbert A. Simon. The architecture of complexity. *Proceedings of the American Philosophical Society*, 106(6):467–482, 1962.

Author Index

Argente, E. 74
Armano, Giuliano 1

Baldoni, Matteo 14
Behrens, Tristan 133
Boella, Guido 14
Botia, Juan 228
Botti, V. 74
Brandsema, Jaap 44
Braubach, Lars 29

Cadenas, Jose Manuel 228
Carrascosa, C. 74
Chopra, Amit 148
Collier, Rem 59

Dastani, Mehdi 44
Doan Van Bien, Dinh 59
Dubel, Amco 44

Furbach, Ulrich 89

Genovese, Valerio 14
Giret, A. 74
Grenna, Roberto 14

Hindriks, Koen 163

JuliÃn, V. 74

Lesperance, Yves 118
Lillis, David 59
Logan, Brian 194
Lorini, Emiliano 179

Madden, Neil 194
Meyer, John-Jules 44
Mohammed, Ammar 89
Mugnaini, Andrea 14

NovÃk, Peter 103

Piunti, Michele 179
Pokahr, Alexander 29

Rebollo, M. 74
Ricci, Alessandro 209

Sardina, Sebastian 118
Serrano, Emilio 228
Singh, Munindar 148

van der Torre, Leon 14
van Riemsdijk, M. Birna 163
Vargiu, Eloisa 1
Viroli, Mirko 209

Keyword Index

Agent development tools . . . 1
Agent programming language . . . 194
agent programming languages . . . 103
agent-oriented programming . . . 118
Applications . . . 1
Architecture . . . 148
architecture . . . 74
Artifacts . . . 209

BDI agents . . . 118
BDI Agents . . . 179
BDI model . . . 29
BDI-based Agent Programming . . . 44
Behavioural State Machines . . . 103
Belief Update . . . 179

CArtAgO . . . 209
Cognitive modeling . . . 179
Commitments . . . 148
communication . . . 163
complex systems . . . 228
computer games . . . 133
constraint logic programming (CLP) 89
conversation . . . 163

Datalog . . . 194
debugging . . . 59
Debugging Multi-Agent Programs . 44
Deployed MAS . . . 1
dynamic MAS . . . 133

Environment Programming . . . 209
Epistemic Reasoning . . . 179

Formal Model . . . 209

goals and long-term behaviour . . . 29

heuristics . . . 103

Interoperation . . . 148

Jade . . . 14
Jason . . . 194,209
Jazzyk . . . 103

mental model . . . 163
Middleware . . . 148
modeling . . . 89
Modularity . . . 194
Multi-agent based simulation . . . 228
Multi-Agent Programming Tools . . 44
multi-agent systems . . . 59
Multi-agent systems . . . 228
multi-agent systems programm . . . 133

Organizations . . . 14

Patterns . . . 148
Players . . . 14
powerJade . . . 14
probabilistic action selection . . . 103
profiling . . . 59
PRS architecture . . . 29

rational agent . . . 163
reasoning about action and change 118
Relevance . . . 179
Roles . . . 14

services . . . 74
situation calculus . . . 118
synchronization . . . 163

verification . . . 89
virtual organizations . . . 74
visualisation . . . 59