LNAI 3346

Rafael H. Bordini
Mehdi Dastani
Jürgen Dix
Amal El Fallah Seghrouchni (Eds.)

Programming Multi-Agent Systems

Second International Workshop ProMAS 2004
New York, NY, USA, July 2004
Selected Revised and Invited Papers

Springer

Lecture Notes in Artificial Intelligence 3346

Edited by J. G. Carbonell and J. Siekmann

Subseries of Lecture Notes in Computer Science

Rafael H. Bordini Mehdi Dastani
Jürgen Dix Amal El Fallah Seghrouchni (Eds.)

Programming Multi-Agent Systems

Second International Workshop ProMAS 2004
New York, NY, USA, July 20, 2004
Selected Revised and Invited Papers

Springer

Series Editors

Jaime G. Carbonell, Carnegie Mellon University, Pittsburgh, PA, USA
Jörg Siekmann, University of Saarland, Saarbrücken, Germany

Volume Editors

Rafael H. Bordini
University of Durham, Department of Computer Science
South Road, Durham, DH1 3LE, UK
E-mail: R.Bordini@durham.ac.uk

Mehdi Dastani
Utrecht University, Intelligent Systems Group
PO Box 80.089, 3508 TB Utrecht, The Netherlands
E-mail: mehdi@cs.uu.nl

Jürgen Dix
Technische Universität Clausthal, Institut für Informatik
Julius-Albert-Str. 4, 38678 Clausthal-Zellerfeld, Germany
E-mail: dix@tu-clausthal.de

Amal El Fallah Seghrouchni
University of Paris X, LIP 6
8, Rue du Capitaine Scott, 75015 Paris, France
E-mail: Amal.Elfallah@lip6.fr

Library of Congress Control Number: 2004118421

CR Subject Classification (1998): I.2.11, I.2, C.2.4, D.2, F.3, D.3

ISSN 0302-9743
ISBN 3-540-24559-6 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11386131 06/3142 5 4 3 2 1 0

Preface

These are the proceedings of the 2nd International Workshop on Programming Multi-agent Systems (ProMAS 2004), held in July 2004 in New York (USA) as an associated event of AAMAS 2004, the main international conference dedicated to autonomous agents and multi-agent systems.

The idea of organizing such an event was discussed during the Dagstuhl seminar *Programming Multi-agent Systems Based on Logic* (see [2]), where the focus was on *logic-based approaches*. It was felt that the scope should be broadened beyond logic-based approaches, and thus ProMAS came into being (see [1] for the proceedings of the first event, ProMAS 2003).

Meanwhile, a Steering Committee (Rafael Bordini, Mehdi Dastani, Jürgen Dix, Amal El Fallah Seghrouchni) as well as an AgentLink III Technical Forum Group on *Programming Multi-agent Systems* were established (the latter one was founded on 30 June/1 July 2004 in Rome, Italy (see http:// www.cs.uu.nl/ mehdi/al3tf8.html). Moreover, a Kluwer book on the same topic is underway (to appear early in 2005) and the third workshop ProMAS 2005 will be organized within AAMAS 2005 (see http://www.cs.uu.nl/ProMAS/ for up-to-date information about ProMAS).

One of the driving motivations behind this workshop series is the observation that the area of autonomous agents and multi-agent systems (MAS) has grown into a promising technology offering sensible alternatives for the design of distributed, intelligent systems. Several efforts have been made by academic researchers, by industrialists, and by several standardization consortia in order to provide new tools, methods, and frameworks so as to establish the necessary standards for a wide use of MAS as a technology on its own, not only as a new paradigm.

However, until recently the main focus of the MAS community has been on the development, sometimes by formal methods but often informally, of *concepts* (concerning both mental and social attitudes), *architectures*, *coordination techniques*, and general approaches to the *analysis and specification* of multi-agent systems. In particular, this contribution has been quite fragmented, without any clear way of "putting it all together," and thus completely inaccessible to practitioners.

We are convinced that the next step in furthering the achievement of the MAS project is irrevocably associated with the *development of programming languages and tools that can effectively support MAS programming* and the *implementation of key notions in multi-agent systems in a unified framework*. The success of agent-oriented system development can only be guaranteed if we can bridge the gap between analysis and design on the one hand, and implementation on the other hand. This, in turn, requires the development of powerful and general-purpose programming technology such that the concepts and techniques of multi-agent systems can be easily and directly implemented.

ProMAS 2004, as indeed ProMAS 2003, was an invaluable opportunity that brought together leading researchers from both academia and industry to discuss the design of programming languages and tools for multi-agent systems. In particular, the workshop promoted the discussion and exchange of ideas concerning the concepts, properties, requirements, and principles that are important for future programming technology for multi-agent systems.

This volume of the LNAI series constitutes the official (post-)proceedings of ProMAS 2004. It presents the main contributions that featured in the latest ProMAS event. Besides the final 10 high-quality accepted papers, we also invited two leading researchers, Milind Tambe and David Kinny, in academia and industry, respectively, to give invited talks at the workshop. Subsequently, they wrote invited contributions which are featured in these proceedings.

The main topics addressed in this volume are:

Agent-Oriented Programming: The first paper in this part of the proceedings, *Goal Representation for BDI Agent Systems*, by Lars Braubach, Alexander Pokahr, and Daniel Moldt, describes a goal model that shows how an agent achieves and manages his goals. It goes on to provide a generic life cycle that models different types of goals in BDI agent systems.

The second paper, *AF-APL – Bridging Principles & Practice in Agent Oriented Languages*, by Robert Ross, Rem Collier, and Gregory M.P. O'Hare, presents an agent-oriented programming language called Agent-Factory. The theoretical foundations of this language are based on principles from agent-oriented design that are enriched with practical considerations of programming real-world agents.

Agent Platforms and Tools: The first paper in this part is *A Toolkit for the Realization of Constraint-Based Multiagent Systems*, by Federico Bergenti. The paper has two main contributions: the first consists of an approach for modelling and a language (called QPL) for programming multi-agent systems where agents are seen as solvers of constraint satisfaction and optimization problems; the second, more practical contribution is the QK toolkit which provides a QPL compiler and a runtime platform for deploying such (constraint-based) multi-agent systems.

The next paper, *Debugging Agent Behavior in an Implemented Agent System*, by Dung Lam and Suzanne Barber, introduces a working tool for debugging BDI multi-agent systems (a research topic that should receive much attention in the future). As the tool is aimed for use with any agent platforms, users have to add their own code for logging run-time data on agents' attitudes; the approach also requires users to provide (usually domain-dependent) rules that relate agent attitudes. Such rules are used by the system to generate interpretations of observed behavior, which can then be compared to a given specification of expected behavior.

The final paper in this part is *A Mobile Agents Platform: Architecture, Mobility and Security Elements*, by Alexandru Suna and Amal El Fallah Seghrouchni. It presents the SyMPA platform for the execution of agents implemented in a high-level declarative agent-oriented programming language called CLAIM, which supports also mobile agents. CLAIM is inspired by ideas from both agent-oriented programming and the Ambient Calculus. SyMPA provides mechanisms for both strong and weak mobility, various aspects of security, and fault tolerance.

Agent Languages: In this part, two papers discussing the implementation of communication models in multi-agent systems are included. The first paper, *Bridging the Gap Between AUML and Implementation Using IOM/T*, by Takuo Doi, Nobukazu Yoshioka, Yasuyuki Tahara, and Shinichi Honiden, presents an interaction protocol description language called IOM/T. This language has a clear correspondence with AUML diagrams and helps to bridge the gap between design and implementation since IOM/T code can be directly converted into Java.

The second paper, *Inter-agent Communication in IMAGO Prolog*, by Xining Li and Guillaume Autran, presents a communication model for a variant of Prolog called IMAGO Prolog. In this model, agent communication is implemented by mobile messengers, which are simple mobile agents that carry messages between agents.

Multi-agent Systems Techniques: The first paper in this part, *OMNI: Introducing Social Structure, Norms and Ontologies into Agent Organizations*, by Virginia Dignum, Javier Vazquez-Salceda, and Frank Dignum, describes a framework for modelling Agent organizations. An important point is to balance both global organizational requirements as well as autonomy of individual agents. Several levels of abstraction are distinguished, each with a formal logical semantics.

The second paper, *A Dialogue Game to Offer an Agreement to Disagree*, by Henk-Jan Lebbink, Cilia Witteman, John-Jules Ch. Meyer, deals with the problem of deciding whether several agents may reach an agreement or not. A particular game is described that allows agents to come to an agreement to disagree and thus to conclude an ongoing dialogue.

The last paper in this part, *Coordination of Complex Systems Based on Multi-agent planning: Application to the Aircraft Simulation Domain*, by Frederic Marc, Amal El Fallah Seghrouchni, and Irene Degirmenciyan-Cartault, is concerned with multi-agent planning in the tactical aircraft simulation domain.

In addition to the peer-reviewed papers listed above, the proceedings contain two invited papers related to the two invited talks given at ProMAS 2004:

– The first invited paper, *Coordinating Teams in Uncertain Environments: A Hybrid BDI-POMDP Approach*, by Ranjit Nair and Milind Tambe, addresses the issue of multi-agent team coordination in the context of multi-agent planning under uncertainty. The paper overviews coordination approaches based on POMDP (Partially Observable Markov Decision Processes) and discusses the limits of POMDP-based approaches in terms of tractability. Then, the authors introduce a hybrid approach to improve the tractability of the POMDP technique. The proposed approach combines two paradigms to build multi-agent team plans: the BDI model (beliefs, desires and intentions) and distributed POMDP. It shows how the use of BDI techniques can improve distributed POMDP and how, reciprocally, distributed POMDP can improve planning performance.

The contributions of the hybrid approach put forward by the authors are: (i) it focuses on agents' roles and on their allocation in teams while taking into account future uncertainties in the studied domain; (ii) it provides a new decomposition technique that exploits the structure of the BDI team plans in order to prune the search space of combinatorial role allocations; and (iii) it proposes a faster policy evaluation algorithm suited to the proposed BDI-POMDP hybrid approach. The paper also presents experimental results from two significant domains: mission rehearsal simulation and the RoboCup Rescue disaster rescue simulation.

– The second invited paper, *Agents – the Challenge of Relevance to the IT Mainstream*, by David Kinny, discusses and evaluates the state of the art in multi-agent systems research by focusing on its relevance to enterprise computing. In particular, it poses the question of whether the agent paradigm and its various technologies are relevant to mainstream IT. To answer this question, the author points out some reasons why other paradigms that promised to transform software development have failed to be adopted by mainstream IT, and explains why and in which ways the agent paradigm has a better prospect of being adopted by the mainstream software industry. It is argued that the agent paradigm will be adopted by mainstream IT if it provides effective solutions to enterprise needs, and delivers substantial benefits that cannot be achieved by other paradigms. The paper presents some valuable aspects of agent technology for enterprise computing and indicates agent techniques and technologies that are ready for mainstream use. To conclude, some of the current research challenges in furthering the relevance of agent technology to mainstream IT are discussed.

The workshop finished with a panel session, moderated by Andrea Omicini, on *Current Trends and Future Challenges in Programming Multi-agent Systems.* The panelists, including again researchers and developers from both academia and industry, were: Monique Calisti, David Kinny, Michael Luck, Onn Shehory, and Franco Zambonelli. Besides various important remarks about how to foster the industrial take-up of MAS technology — such as the general indication that industry seems to be only willing to take small, simple technological advances at a time and in such a way that they can be integrated easily with their existing practices, and the need for working tools (for testing and debugging as well as programming) to support the activities of programmers in industry — a considerable part of the panel session was dedicated to discussing an essential mechanism for technological transfer which is often neglected: the point was that we need to recognize, and exploit sensibly the immense power that academics have in educating the next generations of programmers, who in turn shape the actual practice of the software industry.

We would like to thank all the authors, invited speakers, Programme Committee members, and reviewers for their outstanding contribution to the success of ProMAS 2004. We are especially thankful to the AAMAS 2004 organizers (in particular Simon Parsons) for their technical support and for hosting ProMAS 2004.

October 2004

Rafael H. Bordini
Mehdi Dastani
Jürgen Dix
Amal El Fallah Seghrouchni

References

1. Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors. *Programming Multi-agent Systems (ProMAS 2003)*, LNCS 3067, Berlin, 2004. Springer.
2. Jürgen Dix, Michael Fisher, and Yingqian Zhang. Programming Multi-agent Systems Based on Logic. Technical Report, Dagstuhl Seminar Report 361, IBFI GmbH, Schloß Dagstuhl, 2002.

Organization

ProMAS 2004 was held as a workshop of the 3rd International Joint Conference on Autonomous Agents and Multi-agent Systems, in New York City, NY, on the 20th of July 2004.

Organizing Committee

Rafael H. Bordini (University of Durham, UK)
Mehdi Dastani (Utrecht University, Netherlands)
Jürgen Dix (TU Clausthal, Germany)
Amal El Fallah Seghrouchni (University of Paris VI, France)

Programme Committee

Patrick Albert (Ilog, France)
Rafael H. Bordini (University of Durham, UK)
Jean-Pierre Briot (University of Paris VI, France)
Yves Demazeau (Institut IMAG – Grenoble, France)
Franck Dignum (Utrecht University, Netherlands)
Michael Fisher (University of Liverpool, UK)
Rosa Laura Zavala Gutierrez (University of South Carolina, USA)
Michael Huhns (University of South Carolina, USA)
Toru Ishida (Kyoto University, Japan)
João Alexandre Leite (Universidade Nova de Lisboa, Portugal)
Christian Lemaitre (LANIA, Mexico)
Oliver Obst (Koblenz-Landau University, Germany)
Andrea Omicini (University of Bologna, Italy)
John-Jules Meyer (Utrecht University, Netherlands)
Julian Padget (University of Bath, UK)
Yoav Shoham (Stanford University, USA)
Leendert van der Torre (CWI, Netherlands)
Paolo Torroni (University of Bologna, Italy)
Cees Witteveen (Delft University, Netherlands)

Additional Reviewers

Federico Chesani (University of Bologna, Italy)
Manuela Citrigno (University of Bologna, Italy)
Marc-Philippe Huget (University of Savoy, France)

Table of Contents

V Multi-agent Systems Techniques

Coordinating Teams in Uncertain Environments: A Hybrid BDI-POMDP Approach

Ranjit Nair and Milind Tambe

Computer Science Department,
University of Southern California,
Los Angeles, CA 90089
{nair, tambe}@usc.edu

Abstract. Distributed partially observable Markov decision problems (POMDPs) have emerged as a popular decision-theoretic approach for planning for multiagent teams, where it is imperative for the agents to be able to reason about the rewards (and costs) for their actions in the presence of uncertainty. However, finding the optimal distributed POMDP policy is computationally intractable (NEXP-Complete). This paper is focussed on a principled way to combine the two dominant paradigms for building multiagent team plans, namely the "belief-desire-intention" (BDI) approach and distributed POMDPs. In this hybrid BDI-POMDP approach, BDI team plans are exploited to improve distributed POMDP tractability and distributed POMDP-based analysis improves BDI team plan performance. Concretely, we focus on role allocation, a fundamental problem in BDI teams – which agents to allocate to the different roles in the team. The hybrid BDI-POMDP approach provides three key contributions. First, unlike prior work in multiagent role allocation, we describe a role allocation technique that takes into account future uncertainties in the domain. The second contribution is a novel decomposition technique, which exploits the structure in the BDI team plans to significantly prune the search space of combinatorially many role allocations. Our third key contribution is a significantly faster policy evaluation algorithm suited for our BDI-POMDP hybrid approach. Finally, we also present experimental results from two domains: mission rehearsal simulation and RoboCupRescue disaster rescue simulation. In the RoboCupRescue domain, we show that the role allocation technique presented in this paper is capable of performing at human expert levels by comparing with the allocations chosen by humans in the actual RoboCupRescue simulation environment.

1 Introduction

Teamwork, whether among software agents, or robots (and people) is a critical capability in a large number of multiagent domains ranging from mission rehearsal simulations to RoboCup soccer and disaster rescue to personal assistant teams. Already a large number of multiagent teams have been developed for a range of domains [31, 44, 36, 19, 13, 9, 38, 7]. These existing practical approaches can be characterized as situated within the general "belief-desire-intention" (BDI) approach, a paradigm for designing multiagent systems, made increasingly popular due to programming frameworks [38, 9, 39] that facilitate the design of large-scale teams. Within this approach, inspired explicitly or implicitly by BDI logics, agents explicitly represent and reason with their team goals and plans [41].

This paper focuses on the the quantitative evaluation of multiagent teamwork, to provide feedback to aid human developers and possibly to agents participating in a team, on how the team performance in complex domains can be improved. Such quantitative evaluation is especially vital in domains like disaster rescue [21] and mission rehearsal simulations [38], where the performance of the team is linked to important metrics such as loss of human life and property. Both these and other such complex domains exhibit uncertainty, which arises from partial observability and non-determinism in the outcomes of actions. However, tools for such quantitative evaluations of teamwork in the presence of uncertainty are currently absent. Thus, given these uncertainties, we may be required to experimentally recreate a large number of possible scenarios (in a real domain or in simulations) in order to correctly evaluate team performance.

Fortunately, the emergence of distributed **P**artially **O**bservable **M**arkov **D**ecision **P**roblems (POMDPs) provides models [3, 4, 30, 43] that are well-suited for quantitative analysis of agent teams in uncertain domains. These models are powerful enough to express the uncertainty in these dynamic domains and in principle, can be used to generate and evaluate complete policies for the multiagent team. However, as shown by Bernstein et al. [3], the problem of deriving the optimal policy is generally computationally intractable (the corresponding decision problem is NEXP-complete).

This paper deals with this issue of intractability in distributed POMDPs by combining in a principled way the two dominant paradigms for building multiagent teams, namely distributed POMDPs and the "belief-desire-intention" (BDI) approach. While BDI frameworks facilitate human design of large scale teams, their key shortcoming is their inability to quantitatively reason about team performance, especially in the presence of uncertainty. This hybrid BDI-POMDP approach combines the native strengths of the BDI and POMDP approaches, i.e., the ability in BDI frameworks to encode large-scale team plans and the POMDP ability to quantitatively evaluate such plans. This approach focuses on the analysis of BDI team plans, to provide feedback to human developers on how the team plans can be improved. In particular, it focuses on the critical challenge of role allocation in building teams [40, 18], i.e. which agents to allocate to the various roles in the team. For instance, in mission rehearsal simulations [38], we need to select the numbers and types of helicopter agents to allocate to different roles in the team. Similarly, in disaster rescue [21], role allocation refers to allocating fire engines and ambulances to fires and it can greatly impact team performance. In such domains, the role allocation chosen directly impacts the team performance, which is linked to metrics like loss of human life and property; and thus, it is critical to find the best role allocation.

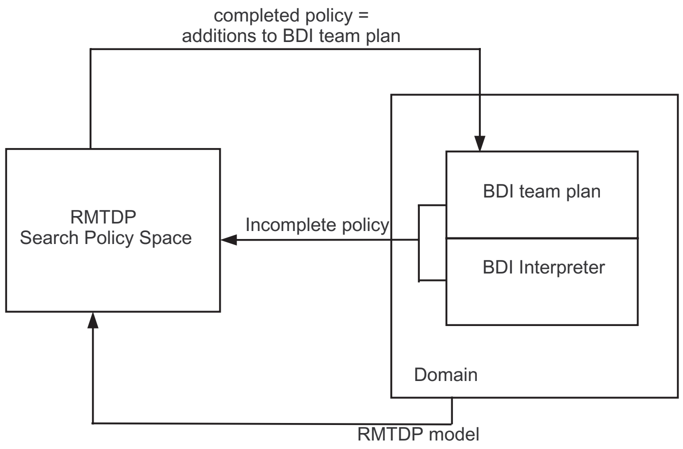

completed policy = additions to BDI team plan
RMTDP Search Policy Space
Incomplete policy
BDI team plan
BDI Interpreter
Domain
RMTDP model

Fig. 1. Integration of BDI and POMDP

In order to analyze role allocations quantitatively, we derive RMTDP (Role-based Multiagent Team Decision Problem), a distributed POMDP framework for quantitatively analyzing role allocations. Using this framework, we show that, in general, the problem of finding the optimal role allocation policy is computationally intractable (the corresponding decision problem is still NEXP-complete). This shows that improving the tractability of analysis techniques for role allocation is a critically important issue.

The hybrid BDI-POMDP approach is based on three key interactions that improve the tractability of RMTDP and the optimality of BDI agent teams. The first interaction is shown in Figure 1. In particular, suppose we wish to analyze a BDI agent team (each agent consisting of a BDI team plan and a domain independent interpreter that helps coordinate such plans) acting in a domain. Then as shown in Figure 1, we model the domain via an RMTDP, and rely on the BDI team plan and interpreter for providing an incomplete policy for this RMTDP. The RMTDP model evaluates different completions of this incomplete policy and provides an optimally completed policy as feedback to the BDI system. Thus, the RMTDP fills in the gaps in an incompletely specified BDI team plan optimally. Here the gaps we concentrate on are the role allocations, but the method can be applied to other key coordination decisions. By restricting the optimization to only role allocation decisions and fixing the policy at all other points, we are able to come up with a restricted policy space. We then use RMTDPs to effectively search this restricted space in order to find the optimal role allocation.

While the restricted policy search is one key positive interaction in our hybrid approach, the second interaction consists of a more efficient policy representation used for converting a BDI team plan and interpreter into a corresponding policy (see Figure 1) and a new algorithm for policy evaluation. In general, each agent's policy in a distributed POMDP is indexed by its observation history [3, 30]. However, in a BDI system, each agent performs its action selection based on its set of privately held beliefs which is obtained from the agent's observations after applying a belief revision function. In order to evaluate the team's performance, it is sufficient in RMTDP to index the agents' policies by their *belief state* (represented here by their privately held beliefs) instead of their observation histories. This shift in representation results in considerable savings in the amount of space and time needed to evaluate a policy.

The third key interaction in our hybrid approach further exploits BDI team plan structure for increasing the efficiency of our RMTDP-based analysis. Even though RMTDP policy space is restricted to filling in gaps in incomplete policies, many policies may result given the large number of possible role allocations. Thus enumerating and evaluating each possible policy for a given domain is difficult. Instead, we provide a branch-and-bound algorithm that exploits task decomposition among sub-teams of a team to significantly prune the search space and provide a correctness proof and worst-case analysis of this algorithm.

In order to empirically validate our approach, we have applied RMTDP for allocation in BDI teams in two concrete domains: mission rehearsal simulations [38] and RoboCupRescue [21]. We first present the (significant) speed-up gained by our three interactions mentioned above. Next, in both domains, we compared the role allocations found by our approach with state-of-the-art techniques that allocate roles without uncertainty reasoning. This comparison shows the importance of reasoning about uncertainty when determining the role allocation for complex multiagent domains. In the RoboCupRescue domain, we also compared the allocations found with allocations chosen by humans in the actual RoboCupRescue simulation environment. The results showed that the role allocation technique presented in this paper is capable of performing at human expert levels in the RoboCupRescue domain.

This paper is organized as follows: In Section 2, background and motivation are presented. In Section 3, we introduce the RMTDP model and present key complexity results. Section 4 explains how a BDI team plan can be evaluated using RMTDP. Section 5 describes the analysis methodology for finding the optimal role allocation, and also presents an empirical evaluation of this methodology. Section 6 describes the JESP approach for finding locally optimal distributed POMDP policies. In Section 7, we present related work, in Section 8, we list our conclusions and in Section 9 we describe future work.

2 Background

This section first describes the two domains that we consider in this paper: an abstract mission rehearsal domain [38] and the RoboCupRescue domain [21]. Each domain requires us to allocate roles to agents in a team. Next, team-oriented programming (TOP), a framework for describing team plans is described in the context of these two domains. While we focus on TOP, as discussed further in Section 7.1, our techniques would be applicable in other frameworks for tasking teams [36, 9].

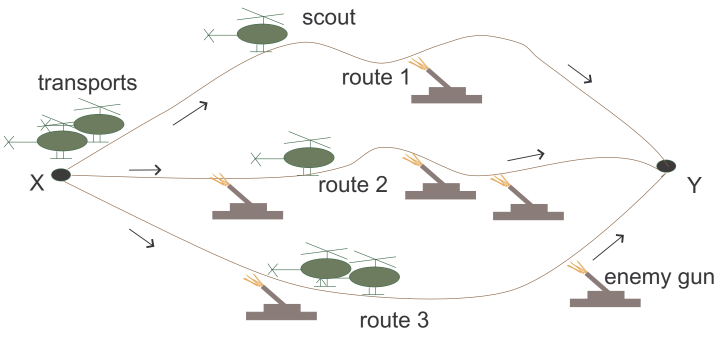


Fig. 2. Mission rehearsal domain

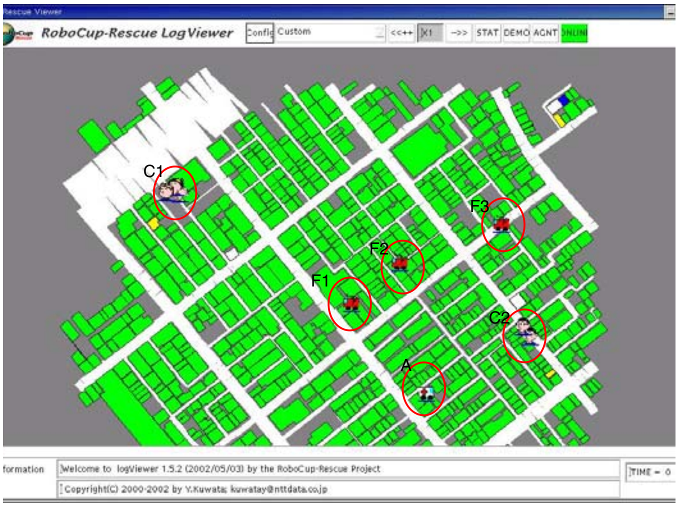
2.1 Domains

The first domain that we consider is based on mission rehearsal simulations [38]. For expository purposes, this has been intentionally simplified. The scenario is as follows: A helicopter team is executing a mission of transporting valuable cargo from point X to point Y through enemy terrain (see Figure 2). There are three paths from X to Y of different lengths and different risk due to enemy fire. One or more scouting sub-teams must be sent out (one for each path from X to Y), and the larger the size of a scouting sub-team the safer it is. When scouts clear up any one path from X to Y, the transports can then move more safely along that path. However, the scouts may fail along a path, and may need to be replaced by a transport at the cost of not transporting cargo. Owing to partial observability, the transports may not receive an observation that a scout has failed or that a route has been cleared. We wish to transport the most amount of cargo in the quickest possible manner within the mission deadline.

The key role allocation decision here is given a fixed number of helicopters, how should they be allocated to scouting and transport roles? Allocating more scouts means that the scouting task is more likely to succeed, but there will be fewer helicopters left that can be used to transport the cargo and consequently less reward. However, allocating too few scouts could result in the mission failing altogether. Also, in allocating the scouts, which routes should the scouts be sent on? The shortest route would be preferable but it is more risky. Sending all the scouts on the same route decreases the likelihood of failure of an individual scout; however, it might be more beneficial to send them on different routes, e.g. some scouts on a risky but short route and others on a safe but longer route.

Thus there are many role allocations to consider. Evaluating each one is difficult because role allocation must look-ahead to consider future implications of uncertainty, e.g. scout helicopters can fail during scouting and may need to be replaced by a transport. Furthermore, failure or success of a scout may not be visible to the transport helicopters and hence a transport may not replace a scout or transports may never fly to the destination.

The second example scenario (see Figure 3), set up in the RoboCupRescue disaster simulation environment [21], consists of five fire engines at three different



Welcome to logViewer 1.5.2 (2002/05/03) by the RoboCup-Rescue Project
Copyright(C) 2000-2002 by Y.Kuwata: kuwatay@nttdata.co.jp

Fig. 3. RoboCupRescue Scenario: C1 and C2 denote the two fire locations, F1, F2 and F3 denote fire stations 1, 2 and 3 respectively and A denotes the ambulance center

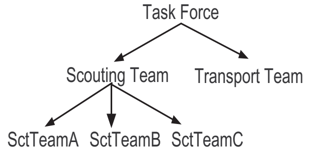
fire stations (two each at stations 1 & 3 and the last one at station 2) and five ambulances stationed at the ambulance center. Two fires (in the top left and bottom right corners of the map) start that need to be extinguished by the fire engines. After a fire is extinguished, ambulance agents need to save the surviving civilians. The number of civilians at each location is not known ahead of time, although the total number of civilians in known. As time passes, there is a high likelihood that the health of civilians will deteriorate and fires will increase in intensity. Yet the agents need to rescue as many civilians as possible with minimal damage to the buildings. The first part of the goal in this scenario is therefore to first determine which fire engines to assign to each fire. Once the fire engines have gathered information about the number of civilians at each fire, this is transmitted to the ambulances. The next part of the goal is then to allocate the ambulances to a particular fire to rescue the civilians trapped there. However, ambulances cannot rescue civilians until fires are fully extinguished. Here, partial observability (each agent can only view objects within its visual range), and uncertainty related to fire intensity, as well as location of civilians and their health add significantly to the difficulty.

2.2 Team-Oriented Programming

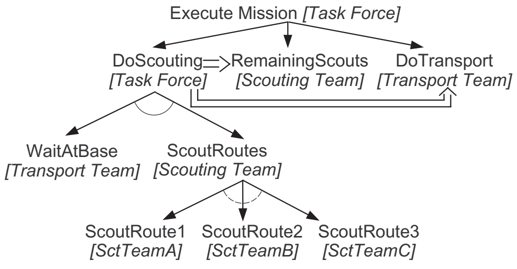
The aim of the team-oriented programming (TOP) [31, 38, 39] framework is to provide human developers (or automated symbolic planners) with a useful abstraction for tasking teams. For domains such as those described in Section 2.1, it consists of three key aspects of a team: (i) a team organization hierarchy consisting of roles; (ii) a team (reactive) plan hierarchy; and (iii) an assignment of roles to sub-plans in the plan hierarchy. The developer need not specify low-level coordination details. Instead, the TOP interpreter (the underlying coordination infrastructure) automatically enables agents to decide when and with whom to communicate and how to reallocate roles upon failure. The TOP abstraction enables humans to rapidly provide team plans for large-scale teams, but unfortunately, only a qualitative assessment of team performance is feasible. Thus, a key TOP weakness is the inability to quantitatively evaluate and optimize team performance. For example, in allocating roles to agents only a qualitative matching of capabilities may be feasible. As discussed later, our hybrid BDI-POMDP model addresses this weakness by providing techniques for quantitative evaluation.

As a concrete example, consider the TOP for the mission rehearsal domain. We first specify the team organization hierarchy (see Figure 4(a)). *Task Force* is the highest level team in this organization and consists of two roles *Scouting* and *Transport*, where the *Scouting* sub-team has roles for each of the three scouting sub-sub-teams. Next we specify a hierarchy of reactive team plans (Figure 4(b)). Reactive team plans explicitly express joint activities of the relevant team and consist of: (i) pre-conditions under which the plan is to be proposed; (ii) termination conditions under which the plan is to be ended; and (iii) team-level actions to be executed as part of the plan (an example plan will be discussed shortly). In Figure 4(b), the highest level plan **Execute Mission** has three sub-plans: **DoScouting** to make one path from X to Y safe for the transports, **DoTransport** to move the transports along a scouted path, and **RemainingScouts** for the scouts which have not reached the destination yet to get there.

Figure 4(b) also shows coordination relationships: An AND relationship is indicated with a solid arc, while an OR relationship is indicated with a dashed arc. Thus, **WaitAtBase** and **ScoutRoutes** must both be done while at least one of **ScoutRoute1**, **ScoutRoute2** or **ScoutRoute3** need be performed. There is also a temporal dependence relationship among the sub-plans, which implies that sub-teams assigned to perform **DoTransport** or **RemainingScouts** cannot do so until the **DoScouting** plan has completed. However, **DoTransport** and



(a)


Execute Mission [Task Force]
DoScouting [Task Force]
RemainingScouts [Scouting Team]
DoTransport [Transport Team]
WaitAtBase [Transport Team]
ScoutRoutes [Scouting Team]
ScoutRoute1 [SctTeamA]
ScoutRoute2 [SctTeamB]
ScoutRoute3 [SctTeamC]

(b)

Fig. 4. TOP for mission rehearsal domain. a) Organization hierarchy; b) Plan hierarchy

```
ExecuteMission:
Context:∅
Pre-conditions: (MB <TaskForce> location(TaskForce) = START)
Achieved: (MB <TaskForce> (Achieved(DoScouting) ∧
          Achieved(DoTransport))) ∧ (time > T ∨ (MB <TaskForce>
          Achieved(RemainingScouts) ∨ (∄ helo ∈ ScoutingTeam,
          alive(helo) ∧ location(helo) ≠ END)))
Unachievable: (MB <TaskForce> Unachievable(DoScouting)) ∨
              (MB <TaskForce> Unachievable(DoTransport) ∧
              (Achieved(RemainingScouts) ∨(∄ helo ∈ ScoutingTeam,
              alive(helo) ∧ location(helo) ≠ END)))
Irrelevant: ∅
Body:
       DoScouting
       DoTransport
       RemainingScouts
Constraints: DoScouting → DoTransport, DoScouting → RemainingScouts
```

Fig. 5. Example team plan. MB refers to mutual belief

RemainingScouts execute in parallel. Finally, we assign roles to plans – Figure 4(b) shows the assignment in brackets adjacent to the plans. For instance, *Task Force* team is assigned to jointly perform **Execute Mission** while *SctTeamA* is assigned to **ScoutRoute1**.

The team plan corresponding to **Execute Mission** is shown in Figure 5. As can be seen, each team plan consists of a context, pre-conditions, post-conditions, body and constraints. The context describes the conditions that must be fulfilled in the parent plan while the pre-conditions are the particular conditions that will cause this sub-plan to begin execution. Thus, for **Execute Mission**, the pre-condition is that the team mutually believes (MB) that they are the "start" location. The post-conditions are divided into Achieved, Unachievable and Irrelevant conditions under which this sub-plan will be terminated. The body consists of sub-plans that exist within this team plan. Lastly, constraints describe any temporal constraints that exist between sub-plans in the body.

During execution, each agent has a copy of the TOP. The agent also maintains a set of private beliefs, which are a set of propositions that the agent believes to be true (see Figure 6). When an agent receives new beliefs, i.e. observations (including communication), the belief update function is used to update its set of privately held beliefs. For instance, upon seeing the last scout crashed, a transport may update its privately held beliefs to include the belief "CriticalFailure(DoScouting)". In practical BDI systems, such belief update computation is of low complexity (e.g. constant or linear time). Once beliefs are updated, an agent selects which plan to execute by matching its beliefs with the pre-conditions in the plans. The basic execution cycle is similar to standard reactive planning systems such as PRS [12].

During team plan execution, observations, in the form of communications, often arise because of the coordination actions executed by the TOP interpreter.

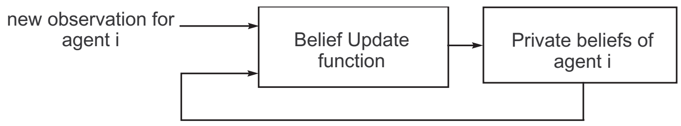


Fig. 6. Mapping of observations to beliefs

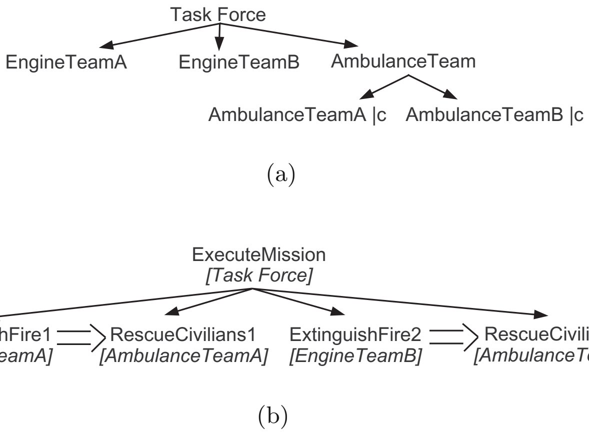


Fig. 7. TOP for RoboCupRescue scenario. a) Organization hierarchy; b) Plan hierarchy

For instance, TOP interpreters have exploited BDI theories of teamwork, such as Levesque et al.'s theory of joint intentions [22], which require that when an agent comes to privately believe a fact that terminates the current team plan (i.e. matches the achievement or unachievability conditions of a team plan), then it communicates this fact to the rest of the team. By performing such coordination actions automatically, the TOP interpreter enables coherence at the initiation and termination of team plans within a TOP. Some further details and examples of TOPs can be seen in [31, 38, 39].

Figure 7 shows the TOP for the RoboCupRescue scenario. As can be seen, the plan hierarchy for this scenario consists of a pair of **ExtinguishFire** and **RescueCivilians** plans done in parallel, each of which further decomposes into individual plans. (These individual plans get the fire engines and ambulances to move through the streets using specific search algorithms. However, these individual plans are not relevant for our discussions in this paper; interested readers should refer to the description of our RoboCupRescue team entered into the RoboCup competitions of 2001 [25].) The organizational hierarchy consists of *Task Force* comprising of two *Engine* sub-teams, one for each fire and an *Ambulance Team*, where the engine teams are assigned to extinguishing the fires while the ambulance team is assigned to rescuing civilians. In this particular TOP, the assignment of ambulances to *AmbulanceTeamA* and *AmbulanceTeamB* is conditioned on the communication "c". For instance, "AmbulanceTeamA|c" is the allocation of ambulances to *AmbulanceTeamA* on receiving communication "c" from the fire engines that describes the number of civilians present at each fire. The problem is which engines to assign to each *Engine Team* and for each possible value of "c", which ambulances to assign to each *Ambulance Team.* Note that engines have differing capabilities owing to differing distances from fires while all the ambulances have identical capabilities.

3 Role-Based Multiagent Team Decision Problem

In order to do quantitative analysis of key coordination decisions in multiagent teams, we extend Multiagent Team Decision Problem (MTDP) [30] for the analysis of the coordination actions of interest. For example, the COM-MTDP [30] is an extension of MTDP for the analysis of communication. In this paper, we illustrate a general methodology for analysis of other aspects of coordination and present the RMTDP model for quantitative analysis of role allocation and reallocation as a concrete example. In contrast to BDI systems introduced in the previous section, RMTDP enables explicit quantitative optimization of team performance. Note that, while we use MTDP, other possible distributed POMDP models could potentially also serve as a basis [3, 43].

3.1 Multiagent Team Decision Problem

Given a team of n agents, an MTDP [30] is defined as a tuple: $\langle S, A, P, \Omega, O, R\rangle$. It consists of a finite set of states $S = \Xi_1 \times \cdots \times \Xi_m$ where each Ξ_j, $1 \leq j \leq m$, is a feature of the world state. Each agent i can perform an action from its set of actions A_i, where $\times_{1\leq i\leq n} A_i = A$. $P(s, < a_1, \ldots, a_n >, s')$ gives the probability of transitioning from state s to state s' given that the agents perform the actions $< a_1, \ldots, a_n >$ jointly. Each agent i receives an observation $\omega_i \in \Omega_i$ ($\times_{1\leq i\leq n} \Omega_i = \Omega$) based on the function $O(s, < a_1, \ldots, a_n >, \omega_1, \ldots, \omega_n)$, which gives the probability that the agents receive the observations, $\omega_1, \ldots, \omega_n$ given that the world state is s and they perform $< a_1, \ldots, a_n >$ jointly. The agents receive a single joint reward $R(s, < a_1, \ldots, a_n >)$ based on the state s and their joint action $< a_1, \ldots, a_n >$. This joint reward is shared equally by all members and there is no other private reward that individual agents receive for their actions. Thus, the agents are motivated to behave as a team, taking the actions that jointly yield the maximum expected reward.

Each agent i in an MTDP chooses its actions based on its local *policy*, π_i, which is a mapping of its observation history to actions. Thus, at time t, agent i will perform action $\pi_i(\omega_i^0, \ldots, \omega_i^t)$. This contrasts with a single-agent POMDP, where we can index an agent's policy by its *belief state* – a probability distribution over the world state [20], which is shown to be a *sufficient statistic* in order to compute the optimal policy [35]. Unfortunately, we cannot directly use single-agent POMDP techniques [20] for maintaining or updating belief states [20] in a MTDP – unlike in a single agent POMDP, in MTDP, an agent's observation depends not only on its own actions, but also on unknown actions of other agents. Thus, as with other distributed POMDP models [3, 43], in MTDP, local policies π_i are indexed by observation histories. $\pi =< \pi_1, \ldots, \pi_n >$ refers to the joint policy of the team of agents.

3.2 Extension for Explicit Coordination

Beginning with MTDP, the next step in our methodology is to make an explicit separation between domain-level actions and the coordination actions of interest. Earlier work introduced the COM-MTDP model [30], where the coordination action was fixed to be the communication action, and got separated out. However, other coordination actions could also be separated from domain-level actions in order to investigate their impact. Thus, to investigate role allocation and reallocations, actions for allocating agents to roles and to reallocate such roles are separated out. To that end, we define RMTDP (Role-based Multiagent Team Decision Problem) as a tuple $\langle S, A, P, \Omega, O, R, \mathcal{RL} \rangle$ with a new component, $\mathcal{RL}$. In particular, $\mathcal{RL} = \{r_1, \ldots, r_s\}$ is a set of all roles that the agents can undertake. Each instance of role r_j may be assigned some agent i to fulfill it. The actions of each agent are now distinguishable into two types:

Role-Taking actions: $\Upsilon_i = \{\upsilon_{ir_j}\}$ contains the role-taking actions for agent i. $\upsilon_{ir_j} \in \Upsilon_i$ means that agent i takes on the role $r_j \in \mathcal{RL}$.

Role-Execution Actions: $\Phi_i = \bigcup_{\forall r_j \in \mathcal{RL}} \Phi_{ir_j}$ contains the execution actions for agent i where Φ_{ir_j} is the set of agent i's actions for executing role $r_j \in \mathcal{RL}$

In addition we define the set of states as $S = \Xi_1 \times \cdots \times \Xi_m \times \Xi_{roles}$, where the feature Ξ_{roles} (a vector) gives the current role that each agent has taken on. The reason for introducing this new feature is to assist us in the mapping from a BDI team plan to an RMTDP. Thus each time an agent performs a new role-taking action successfully, the value of the feature Ξ_{roles} will be updated to reflect this change. The key here is that we not only model an agent's initial role-taking action but also subsequent role reallocation. Modeling both allocation and reallocation is important for an accurate analysis of BDI teams. Note that an agent can observe the part of this feature pertaining to its own current role but it may not observe the parts pertaining to other agents' roles.

The introduction of roles allows us to represent the specialized behaviors associated with each role, e.g. a transport vs. a scout role. While filling a particular role, r_j, agent i can perform only role-execution actions, $\phi \in \Phi_{ir_j}$, which may be different from the role-execution actions Φ_{ir_l} for role r_l. Thus, the feature Ξ_{roles} is used to filter actions such that only those role-execution actions that correspond to the agent's current role are permitted. In the worst case, this filtering does not affect the computational complexity but in practice, it can significantly improve performance when trying to find the optimal policy for the team, since the number of domain actions that each agent can choose from is restricted by the role that the agent has taken on. Also, these different roles can produce varied effects on the world state (modeled via transition probabilities, P) and the team's reward. Thus, the policies must ensure that agents for each role have the capabilities that benefit the team the most.

Just as in MTDP, each agent chooses which action to perform by indexing its local policy π_i by its observation history. In the same epoch some agents could be doing role-taking actions while others are doing role-execution actions. Thus, each agent's local policy π_i can be divided into local role-taking and role-execution policies such that for all observation histories, $\omega_i^0, \ldots, \omega_i^t$, either $\pi_{i\Upsilon}(\omega_i^0, \ldots, \omega_i^t) = \textbf{null}$ or $\pi_{i\Phi}(\omega_i^0, \ldots, \omega_i^t) = \textbf{null}$. $\pi_\Upsilon =< \pi_{1\Upsilon}, \ldots, \pi_{n\Upsilon} >$ refers to the joint role-taking policy of the team of agents while $\pi_\Phi =< \pi_{1\Phi}, \ldots, \pi_{n\Phi} >$ refers to the joint role-execution policy. In this paper, we do not explicitly model communicative actions as a special action. Thus communication is treated like any other role-execution action and the communication received from other agents are treated as observations.

Solving the RMTDP for the optimal role-taking policy, assuming that the role-execution policy is fixed, is highly intractable [28]. In general the globally optimal role-taking policy will be of doubly exponential complexity, and so we may be left no choice but to run a brute-force policy search, i.e. to enumerate all the role-taking policies and then evaluate them, which together determine the run-time of finding the globally optimal policy. The number of policies is $\left(|\Upsilon|^{\frac{|\Omega|^T-1}{|\Omega|-1}}\right)^n$, i.e. doubly exponential in the number of observation histories and the number of agents. Thus, while RMTDP enables quantitative evaluation of team's policies, computing optimal policies is intractable; furthermore, given its low level of abstraction, in contrast to TOP, it is difficult for a human to understand the optimal policy. This contrast between RMTDP and TOP is at the root of our hybrid model described in the following section.

4 Hybrid BDI-POMDP Approach

Having explained TOP and RMTDP, we can now present a more detailed view of our hybrid methodology to quantitatively evaluate a TOP with the help of Figure 1. We first provide a more detailed interpretation of Figure 1. BDI team plans are essentially TOP plans, while the BDI interpreter is the TOP coordination layer. As shown in Figure 1, an RMTDP model is constructed corresponding to the domain and the TOP and its interpreter are converted into a corresponding (incomplete) RMTDP policy. We can then analyze the TOP using analysis techniques that rely on evaluating the RMTDP policy using the RMTDP model of the domain.

Thus, our hybrid approach combines the strengths of the TOPs (enabling humans to specify TOPs to coordinate large-scale teams) with the strengths of RMTDP (enabling quantitative evaluation of different role allocations). On the one hand, this synergistic interaction enable RMTDPs to improve the performance of TOP-based BDI teams. On the other hand, we have identified at least six specific ways in which TOPs make it easier to build RMTDPs and to efficiently search RMTDP policies: two of which are discussed in this section, and four in the next section. In particular, the six ways are:

1. TOPs are exploited in constructing RMTDP models of the domain (Section 4.1);
2. TOPs are exploited to present incomplete policies to RMTDPs, thus restricting the RMTDP policy search (Section 5.1);

3. TOP belief representation is exploited in enabling faster RMTDP policy evaluation (Section 4.2);
4. TOP organization hierarchy is exploited in hierarchically grouping RMTDP policies (Section 5.1);
5. TOP plan hierarchy is exploited in decomposing RMTDPs (Section 5.2);
6. TOP plan hierarchies are also exploited in cutting down the observation or belief histories in RMTDPs (Section 5.2).

The end result of this efficient policy search is a completed RMTDP policy that improves TOP performance. While we exploit the TOP framework, other frameworks for tasking teams, e.g. Decker and Lesser [9] and Stone and Veloso [36] could benefit from a similar synergistic interaction.

4.1 Guidelines for Constructing an RMTDP

As shown in Figure 1, our analysis approach uses as input an RMTDP model of the domain, as well as an incomplete RMTDP policy. Fortunately, not only does the TOP serve as a direct mapping to the RMTDP policy, but it can also be utilized in actually constructing the RMTDP model of the domain. In particular, the TOP can be used to determine which domain features are important to model. In addition, the structure in the TOP can be exploited in decomposing the construction of the RMTDP.

The elements of the RMTDP tuple, $\langle S, A, P, \Omega, O, R, \mathcal{RL} \rangle$, can be defined using a procedure that relies on both the TOP as well as the underlying domain. While this procedure is not automated, our key contribution is recognizing the exploitation of TOP structures in constructing the RMTDP model. First, in order to determine the set of states, S, it is critical to model the variables tested in the pre-conditions, termination conditions and context of all the components (i.e. sub-plans) in the TOP. Note that a state only needs to model the features tested in the TOP; if a TOP pre-condition expresses a complex test on the feature, that test is not modeled in the state, but instead gets used in defining the incomplete policy input to RMTDP. Next we define the set of roles, $\mathcal{RL}$, as the leaf-level roles in the organization hierarchy of the TOP. Furthermore, as specified in Section 3.2, we define a state feature Ξ_{roles} as a vector containing the current role for each agent. Having defined $\mathcal{RL}$ and Ξ_{roles}, we now define the actions, A as follows. For each role $r_j \in \mathcal{RL}$, we define a corresponding role-taking action, υ_{ir_j} which will succeed or fail depending on the agent i that performs the action and the state s that the action was performed in. The role-execution actions, Φ_{ir_j} for agent i in role r_j, are those allowed for that role according to the TOP.

Thus, we have defined S, A and $\mathcal{RL}$ based on the TOP. To illustrate these steps, consider the plans in Figure 4(b). The pre-conditions of the leaf-level plan **ScoutRoute1**, for instance, tests start location of the helicopters to be at start location X, while the termination conditions test that scouts are at end location Y. Thus, the locations of the helicopters are modeled as features in the set of states in the RMTDP. Using the organization hierarchy, we define the set of roles $\mathcal{RL}$ with a role corresponding to each of the four different kinds of leaf-level roles, i.e. $\mathcal{RL} = \{memberSctTeamA, memberSctTeamB, memberSctTeamC, memberTransportTeam\}$. Using these roles, we can define the role-taking and role-execution actions as follows:

- A role-taking action is defined corresponding to each of the four roles in $\mathcal{RL}$, i.e. becoming a member of one of the three scouting teams or of the transport team. The domain specifies that only a transport can change to a scout and thus the role-taking action, *jointTransportTeam*, will fail for agent i, if the current role of agent i is a scout.
- Role-execution actions are obtained from the TOP plans corresponding to the agent's role. In the mission rehearsal scenario, an agent, fulfilling a scout role (members of SctTeamA, SctTeamB or SctTeamC), always goes forward, making the current position safe, until it reaches the destination and so the only execution action we will consider is "move-making-safe". An agent in a transport role (members of Transport Team) waits at X until it obtains observation of a signal that one scouting sub-team has reached Y and hence the role-execution actions are "wait" and "move-forward".

We must now define Ω, P, O, R. We obtain the set of observations Ω_i for each agent i directly from the domain. For instance, the transport helos may observe the status of scout helos (normal or destroyed), as well as a signal that a path is safe. Finally, determining the functions, P, O, R requires some combination of human domain expertise and empirical data on the domain behavior. However, as shown later in Section 6, even an approximate model of transitional and observational uncertainty is sufficient to deliver significant benefits. Defining the reward and transition function may sometimes require additional state variables to be modeled, if they were only implicitly modeled in the TOP. In the mission rehearsal domain, the time at which the scouting and transport mission were completed determined the amount of reward. Thus, time was only implicitly modeled in the TOP and needed to be explicitly modeled in the RMTDP.

Since we are interested in analyzing a particular TOP with respect to uncertainty, the procedure for constructing an RMTDP model can be simplified by exploiting the hierarchical decomposition of the TOP in order to decompose the construction of the RMTDP model. The high-level components of a TOP often represent plans executed by different sub-teams, which may only loosely interact with each other. Within a component, the sub-team members may exhibit a tight interaction, but our focus is on the "loose coupling" across components, where only the end results of one component feed into another, or the components independently contribute to the team goal. Thus, our procedure for constructing an RMTDP exploits this loose coupling between components of the plan hierarchy in order to build an RMTDP model represented as a combination of smaller RMTDPs (factors). Note that if such decomposition is infeasible, our approach still applies except that the benefits of the hierarchical decomposition will be unavailable.

We classify sibling components as being either parallel or sequentially executed (contains a temporal constraint). Components executed in parallel could be either *independent* or *dependent*. For *independent* components, we can define RMTDPs for each of these components such that the sub-team executing one component cannot affect the transitions, observations and reward obtained by the sub-teams executing the other components. The procedure for determining the elements of the RMTDP tuple for component k, $\langle S_k, A_k, P_k, \Omega_k, O_k, R_k, \mathcal{RL}_k \rangle$, is identical to the procedure described earlier for constructing the overall RMTDP. However, each such component has a smaller set of relevant variables and roles and hence specifying the elements of its corresponding RMTDP is easier.

We can now combine the RMTDPs of the independent components to obtain the RMTDP corresponding to the higher-level component. For a higher level component l, whose child components are independent, the set of states, $S_l = \times_{\forall \Xi_x \in F_{S_l}} \Xi_x$ such that $F_{S_l} = \bigcup_{\forall k \ s.t. \ Child(k,l)=\mathbf{true}} F_{S_k}$ where F_{S_l} and F_{S_k} are the sets of features for the set of states S_l and set of states S_k. A state $s_l \in S_l$ is said to correspond to the state $s_k \in S_k$ if $\forall \Xi_x \in F_{S_k}, s_l[\Xi_x] = s_k[\Xi_x]$, i.e. the state s_l has the same value as state s_k for all features of state s_k. The transition function is defined as follows, $P_l(s'_l, a_l, s_l) = \prod_{\forall k \ s.t. \ Child(k,l)=\mathbf{true}} P_k(s'_k, a_k, s_k)$, where s_l and s'_l of component l corresponds to states s_k and s'_k of component k and a_k is the joint action performed by the sub-team assigned to component k corresponding to the joint action a_l performed by the sub-team assigned to component l. The observation function is defined similarly as $O_l(s_l, a_l, \omega_l) = \prod_{\forall k \ s.t. \ Child(k,l)=\mathbf{true}} O_k(s_k, a_k, \omega_k)$. The reward function is defined as $R_l(s_l, a_l) = \sum_{\forall k \ s.t. \ Child(k,l)=\mathbf{true}} R_k(s_k, a_k)$.

In the case of sequentially executed components (those connected by a temporal constraint), the components are loosely coupled since the end states of the preceding component specify the start states of the succeeding component. Thus, since only one component is active at a time, the transition function is defined as follows, $P_l(s'_l, a_l, s_l) = P_k(s'_k, a_k, s_k)$, where component k is the only active child component, s_k and s'_k represent the states of component k corresponding to states s_l and s'_l of component l and a_k is the joint action performed by the sub-team assigned to component k corresponding to the joint action a_l performed by the sub-team corresponding to component l. Similarly, we can define $O_l(s_l, a_l, \omega_l) = O_k(s_k, a_k, \omega_k)$ and $R_l(s_l, a_l) = R_k(s_k, a_k)$, where k is the only active child component.

Consider the following example from the mission rehearsal domain where components exhibit both sequential dependence and parallel independence. Concretely, **DoScouting** is executed first followed by **DoTransport** and **RemainingScouts**, which are parallel and independent and hence, either **DoScouting** is active or **DoTransport** and **RemainingScouts** are active at any point in the execution. Hence, the transition, observation and reward functions of their parent **Execute Mission** is given by the corresponding functions of either **DoScouting** or by the combination of the corresponding functions of **DoTransport** and **RemainingScouts**.

We use a top-down approach in order to determine how to construct a factored RMTDP from the plan hierarchy, where we replace a particular sub-plan by its constituent sub-plans if they are either independent or sequentially executed. If not, then the RMTDP is defined using that particular sub-plan. This process is applied recursively starting at the root component of the plan hierarchy. As a concrete example, consider again our mission rehearsal simulation domain and the hierarchy illustrated in Figure 4(b). Given the temporal constraints between **DoScouting** and **DoTransport**, and **DoScouting** and **RemainingScouts**, we exploited sequential decomposition, while **DoTransport** and **RemainingScouts** were parallel and independent components. Hence, we can replace **ExecuteMission** by **DoScouting**, **DoTransport** and **RemainingScouts**. We then apply the same process to **DoScouting**. The constituent components of **DoScouting** are neither independent nor sequentially executed and thus **DoScouting** cannot be replaced by its constituent components. Thus, RMTDP for the mission rehearsal domain is comprised of smaller RMTDPs for **DoScouting**, **DoTransport** and **RemainingScouts**.

Thus, using the TOP to identify relevant variables and building a factored RMTDP utilizing the structure of TOP to decompose the construction procedure, reduce the load on the domain expert for model construction. Furthermore, as shown in Section 5.2, this factored model greatly improves the performance of the search for the best role allocation.

4.2 Evaluating RMTDP Policies by Exploiting TOP Beliefs

We now present a technique for exploiting TOPs in speeding up evaluation of RMTDP policies. Before we explain our improvement, we first describe the original algorithm for determining the expected reward of a joint policy, where the local policies of each agent are indexed by its entire observation histories [30, 26]. Here, we obtain an RMTDP policy from a TOP as follows. We obtain $\pi_i(\boldsymbol{\omega}_i^t)$, i.e. the action performed by agent i for each observation history $\boldsymbol{\omega}_i^t$, as the action a performed by the agent i following the TOP when it has a set of privately held beliefs corresponding to the observation history, $\boldsymbol{\omega}_i^t$. We compute the expected reward for the RMTDP policy by projecting the team's execution over all possible branches on different world states and different observations. At each time step, we can compute the expected value of a joint policy, $\pi =< \pi_1, \ldots, \pi_n >$, for a team starting in a given state, s^t, with a given set of past observations, $\boldsymbol{\omega}_1^t, \ldots, \boldsymbol{\omega}_n^t$, as follows:

$$\begin{aligned}
V_\pi^t(s^t, \langle \boldsymbol{\omega}_1^t, \ldots, \boldsymbol{\omega}_n^t \rangle) = R(s^t, \langle \pi_1(\boldsymbol{\omega}_1^t), \ldots, \pi_n(\boldsymbol{\omega}_n^t)\rangle) + \\
\sum_{s^{t+1} \in S} P\left(s^t, \left\langle \pi_1\left(\boldsymbol{\omega}_1^t\right), \ldots, \pi_n\left(\boldsymbol{\omega}_n^t\right)\right\rangle, s^{t+1}\right) \cdot \\
\sum_{\omega_{t+1} \in \Omega} O\left(s^{t+1}, \left\langle \pi_1\left(\boldsymbol{\omega}_1^t\right), \ldots, \pi_n\left(\boldsymbol{\omega}_n^t\right)\right\rangle, \left\langle \omega_1^{t+1}, \ldots, \omega_n^{t+1}\right\rangle\right) \cdot \\
V_\pi^{t+1}\left(s^{t+1}, \left\langle \boldsymbol{\omega}_1^{t+1}, \ldots, \boldsymbol{\omega}_n^{t+1}\right\rangle\right) \qquad (1)
\end{aligned}$$

The expected reward of a joint policy π is given by $V_\pi^0(s^0, < \mathbf{null}, \ldots, \mathbf{null} >)$ where s^0 is the start state. At each time step t, the computation of V_π^t performs a summation over all possible world states and agent observations and so has a time complexity of $O\left(|S| \cdot |\Omega|\right)$. This computation is repeated for all states and all observation histories of length t, i.e. $O\left(|S| \cdot |\Omega|^t\right)$ times. Therefore, given a time horizon T, the overall complexity of this algorithm is $O\left(|S|^2 \cdot |\Omega|^{T+1}\right)$.

As discussed in Section 2.2, in a team-oriented program, each agent's action selection is based on just its currently held private beliefs (note that mutual beliefs are modeled as privately held beliefs about all agents as per footnote 2). A similar technique can be exploited when mapping TOP to an RMTDP policy. Indeed, the evaluation of a RMTDP policy that corresponds to a TOP can be speeded up if each agent's local policy is indexed by its private beliefs, ${\psi_i}^t$. We refer to ${\psi_i}^t$, as the TOP-congruent belief state of agent i in the RMTDP. Note that this belief state is not a probability distribution over the world states as in a single agent POMDP, but rather the privately held beliefs (from the BDI program) of agent i at time t.

Belief-based RMTDP policy evaluation leads to speedup because multiple observation histories map to the same belief state, ${\psi_i}^t$. This speedup is a key illustration of exploitation of synergistic interactions of TOP and RMTDP. In this instance, belief representation techniques used in TOP are reflected in RMTDP, and the resulting faster policy evaluation can help us optimize TOP performance. A detailed example of belief state is presented later after a brief explanation of how such belief-based RMTDP policies can be evaluated.

Just as with evaluation using observation histories, we compute the expected reward of a belief-based policy by projecting the team's execution over all possible branches on different world states and different observations. At each time step, we can compute the expected value of a joint policy, $\pi =< \pi_1, \ldots, \pi_n >$, for a team starting in a given state, s^t, with a given team belief state, $< {\psi_1}^t, \ldots, {\psi_n}^t >$ as follows:

$$\begin{aligned} V_\pi^t(s^t, \langle {\psi_1}^t \ldots {\psi_n}^t \rangle) = {} & R(s^t, \langle \pi_1({\psi_1}^t), \ldots, \pi_n({\psi_n}^t)\rangle) + \\ & \sum_{s^{t+1} \in S} P\left(s^t, \left\langle \pi_1\left({\psi_1}^t\right), \ldots, \pi_n\left({\psi_n}^t\right)\right\rangle, s^{t+1}\right) \cdot \\ & \sum_{\omega_{t+1} \in \Omega} O\left(s^{t+1}, \left\langle \pi_1\left({\psi_1}^t\right), \ldots, \pi_n\left({\psi_n}^t\right)\right\rangle, \left\langle \omega_1^{t+1}, \ldots, \omega_n^{t+1}\right\rangle\right) \cdot \\ & V_\pi^{t+1}\left(s^{t+1}, \left\langle {\psi_1}^{t+1}, \ldots, {\psi_n}^{t+1}\right\rangle\right) \qquad (2) \end{aligned}$$

$$where\ {\psi_i}^{t+1} = \mathbf{BeliefUpdateFunction}\left({\psi_i}^t, \omega_i^{t+1}\right)$$

The complexity of computing this function (expression 2) is $O\left(|S| \cdot |\Omega|\right) \cdot O\left(\mathbf{Belief\ \ UpdateFunction}\right)$. At each time step the computation of the value function is done for every state and for all possible reachable belief states. Let $|\Psi_i| = \mathbf{max}_{1 \leq t \leq T}(|\psi_i^t|)$ represent the maximum number of possible belief states that agent i can be in at any point in time, where $|\psi_i^t|$ is the number of belief states that agent i can be in at t. Therefore the complexity of this algorithm is given by $O(|S|^2 \cdot |\Omega| \cdot (|\Psi_1| \cdot \ldots \cdot |\Psi_n|) \cdot T) \cdot O\left(\mathbf{BeliefUpdateFunction}\right)$. Note that, in this algorithm T is not in the exponent unlike in the algorithm in expression 1. Thus, this evaluation method will give large time savings if: (i) the quantity $(|\Psi_1| \cdot \ldots \cdot |\Psi_n|) \cdot T$ is much less than $|\Omega|^T$ and (ii) the belief update cost is low. In practical BDI systems, multiple observation histories map often onto the same belief state, and thus usually, $(|\Psi_1| \cdot \ldots \cdot |\Psi_n|) \cdot T$ is much less than $|\Omega|^T$. Furthermore, since the belief update function mirrors practical BDI systems, its complexity is also a low polynomial or a constant. Indeed, our experimental results show that significant speedups result from switching to our TOP-congruent belief states ${\psi_i}^t$. However, in the absolute worst case, the belief update function may simply append the new observation to the history of past observations (i.e., TOP-congruent beliefs will be equivalent to keeping entire observation histories) and thus belief-based evaluation will have the same complexity as the observation history-based evaluation.

We now turn to an example of belief-based policy evaluation from the mission rehearsal domain. At each time step, the transport helicopters may receive an observation about whether a scout has failed based on some observation function. If we use the observation-history representation of the policy, then each transport agent would maintain a complete history of the observations that it could receive at each time step. For example, in a setting with two scout helicopters, one on route 1 and the other on route 2, a particular transport helicopter may have several different observation histories of length two. At every time step, the transports may receive an observation about each scout being alive or having failed. Thus, at time $t = 2$, a transport helicopter might have one of the following observation histories of length two, $\left\langle \{sct1OnRoute1Alive, sct2OnRoute2Alive\}^1, \{sct1OnRoute1Failed, sct2OnRoute2Failed\}^2 \right\rangle$, $\left\langle \{sct1OnRoute1Alive, sct2OnRoute2Failed\}^1, \{sct1OnRoute1Failed\}^2 \right\rangle$, $\left\langle \{sct1OnRoute1Failed, sct2OnRoute2Alive\}^1, \{sct2OnRoute2Failed\}^2 \right\rangle$, etc. However, the action selection of the transport helicopters depends on only whether a critical failure (i.e. the last remaining scout has crashed) has taken place to change its role. Whether a failure is critical can be determined by passing each observation through a belief-update function. The exact order in which the observations are received or the precise times at which the failure or non-failure observations are received are not relevant to determining if a critical failure has taken place and consequently whether a transport should change its role to a scout. Thus, many observation histories map onto the same belief states. For example, the above three observation histories all map to the same belief $CriticalFailure(DoScouting)$ i.e. a critical failure has taken place. This results in significant speedups using belief-based evaluation, as Equation 2 needs to be executed over a smaller number of belief states, linear in T in our domains, as opposed to the observation history-based evaluation, where Equation 1 is executed over an exponential number of observation histories ($|\Omega|^T$). The actual speedup obtained in the mission rehearsal domain is demonstrated empirically in Section 6.

5 Optimizing Role Allocation

While Section 4 focused on mapping a domain of interest onto RMTDP and algorithms for policy evaluation, this section focuses on efficient techniques for RMTDP policy search, in service of improving BDI/TOP team plans. The TOP in essence provides an incomplete, fixed policy, and the policy search optimizes decisions left open in the incomplete policy; the policy thus completed optimizes the original TOP (see Figure 1). By enabling the RMTDP to focus its search on incomplete policies, and by providing ready-made decompositions, TOPs assist RMTDPs in quickly searching through the policy space, as illustrated in this section. We focus, in particular, on the problem of role allocation [18, 24, 40, 11], a critical problem in teams. While the TOP provides an incomplete policy, keeping open the role allocation decision for each agent, the RMTDP policy search provides the optimal role-taking action at each of the role allocation decision points. In contrast to previous role allocation approaches, our approach determines the best role allocation, taking into consideration the uncertainty in the domain and future costs. Although demonstrated for solving the role allocation problem, the methodology is general enough to apply to other coordination decisions.

5.1 Hierarchical Grouping of RMTDP Policies

As mentioned earlier, to address role allocation, the TOP provides a policy that is complete, except for the role allocation decisions. RMTDP policy search then optimally fills in the role allocation decisions. To understand the RMTDP policy search, it is useful to gain an understanding of the role allocation search space. First, note that role allocation focuses on deciding how many and what types of agents to allocate to different roles in the organization hierarchy. This role allocation decision may be made at time $t = 0$ or it may be made at a later time conditioned on available observations. Figure 8 shows a partially expanded role allocation space defined by the TOP organization hierarchy in Figure 4(a) for six helicopters. Each node of the role allocation space completely specifies the allocation of agents to roles at the corresponding level of the organization hierarchy (ignore for now, the number to the right of each node). For instance, the root node of the role allocation space specifies that six helicopters are assigned to the *Task Force* (level one) of the organization hierarchy while the leftmost leaf node (at level three) in Figure 8 specifies that one helicopter is assigned to *SctTeamA*, zero to *SctTeamB*, zero to *SctTeamC* and five helicopters to *Transport Team*. Thus, as we can see, each leaf node in the role allocation space is a complete, valid role allocation of agents to roles in the organization hierarchy.

In order to determine if one leaf node (role allocation) is superior to another we evaluate each using the RMTDP by constructing an RMTDP policy for each. In this particular example, the role allocation specified by the leaf node corresponds to the role-taking actions that each agent will execute at time $t = 0$. For example, in the case of the leftmost leaf in Figure 8, at time $t = 0$, one agent (recall from Section 2.2 that this is a homogeneous team and hence which specific agent does not matter) will become a member of *Sct-*

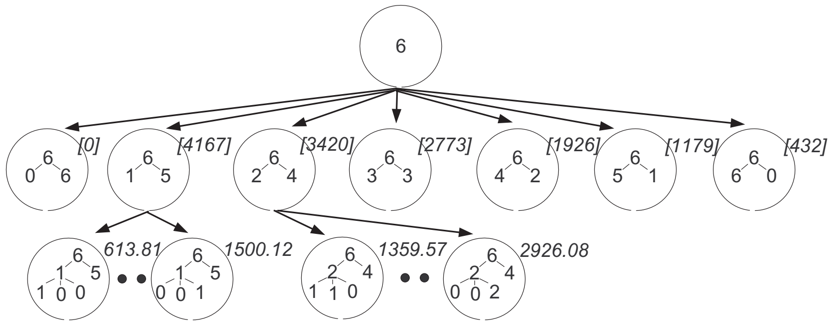


Fig. 8. Partially expanded role allocation space for mission rehearsal domain(six helos)

TeamA while all other agents will become members of *Transport Team*. Thus, for one agent i, the role-taking policy will include $\pi_{i\Upsilon}(\mathbf{null}) = joinSctTeamA$ and for all other agents, $j, j \neq i$, it will include $\pi_{j\Upsilon}(\mathbf{null}) = joinTransportTeam$. In this case, we assume that the rest of the role-taking policy, i.e. how roles will be reallocated if a scout fails, is obtained from the role reallocation algorithm in the BDI/TOP interpreter, such as the *STEAM* algorithm [38]. Thus for example, if the role reallocation is indeed performed by the STEAM algorithm, then STEAM's reallocation policy is included into the incomplete policy that the RMTDP is initially provided. Thus, the best role allocation is computed keeping in mind STEAM's reallocation policy. In STEAM, given a failure of an agent playing $Role_F$, an agent playing $Role_R$ will replace it if:

$$Criticality\,(Role_F) - Criticality\,(Role_R) > 0$$
$$Criticality\,(x) = 1 \;\; if \;\; x \;is\; critical; = 0 \;\; otherwise$$

Thus, if based on the agents' observations, a critical failure has taken place, then the replacing agent's decision to replace or not will be computed using the above expression and then included in the incomplete policy input to the RMTDP. Since such an incomplete policy is completed by the role allocation at each leaf node using the technique above, we have been able to construct a policy for the RMTDP that corresponds to the role allocation.

We are thus able to exploit the TOP organization hierarchy to create a hierarchical grouping of RMTDP policies. In particular, while the leaf node represents a complete RMTDP policy (with the role allocation as specified by the leaf node), a parent node represents a group of policies. Evaluating a policy specified by a leaf node is equivalent to evaluating a specific role allocation while taking future uncertainties into account. We could do a brute force search through all role allocations, evaluating each in order to determine the best role allocation. However, the number of possible role allocations is exponential in the leaf roles in the organization hierarchy. Thus, we must prune the search space.

5.2 Pruning the Role Allocation Space

We prune the space of valid role allocations using upper bounds (MaxEstimates) for the parents of the leaves of the role allocation space as admissible heuristics. Each leaf in the role allocation space represents a completely specified policy and the MaxEstimate is an upper bound of maximum value of all the policies under the same parent node evaluated using the RMTDP. Once we obtain MaxEstimates for all the parent nodes (shown in brackets to the right of each parent node in Figure 8), we use branch-and-bound style pruning. The key novelty of this branch-and-bound algorithm, which we discuss below, is how the we exploit the structure of the TOP in order to compute upper bounds for parent nodes. The other details of this algorithm can be found in [28].

We will now discuss how the upper bounds of parents, called MaxEstimates, can be calculated for each parent. The MaxEstimate of a parent is defined as a strict upper bound of the maximum of the expected reward of all the leaf nodes under it. It is necessary that the MaxEstimate be an upper bound or else we might end up pruning potentially useful role allocations. In order to calculate the MaxEstimate of each parent we could evaluate each of the leaf nodes below it using the RMTDP, but this would nullify the benefit of any subsequent pruning. We, therefore, turn to the TOP plan hierarchy (see Figure 4(b)) to break up this evaluation of the parent node into components, which can be evaluated separately thus decomposing the problem. In other words, our approach exploits the structure of the BDI program to construct small-scale RMTDPs unlike other decomposition techniques which just assume decomposition or ultimately rely on domain experts to identify interactions in the agents' reward and transition functions [8, 15].

For each parent in the role allocation space, we use these small-scale RMTDPs to evaluate the values for each TOP component. Fortunately, as discussed in Section 4.1, we exploited small-scale RMTDPs corresponding to TOP components in constructing larger scale RMTDPs. We put these small-scale RMTDPs to use again, evaluating policies within each component to obtain upper bounds. Note that just like in evaluation of leaf-level policies, the evaluation of components for the parent node can be done using either the observation histories (see Equation 1) or belief states (see Equation 2). We will describe this section using the observation history-based evaluation method for computing the values of the components of each parent, which can be summed up to obtain its MaxEstimate (an upper bound on its children's values). Thus, whereas a parent in the role allocation space represents a group of policies, the TOP components (sub-plans) allow a component-wise evaluation of such a group to obtain an upper bound on the expected reward of any policy within this group.

Algorithm 1 exploits the smaller-scale RMTDP components, discussed in Section 4.1, to obtain upper bounds of parents. First, in order to evaluate the MaxEstimate for each parent node in the role allocation space, we identify the start states for each component from which to evaluate the RMTDPs. We explain this step using a parent node from Figure 8 – *Scouting Team* = two helos, *Transport Team* = four helos (see Figure 9). For the very first component which does not have any preceding components, the start states corresponds to the start states of the policy that the TOP was mapped onto. For each of the next components – where the next component is one linked by a sequential dependence – the start states are the end states of the preceding component. However, as explained later in this section, we can significantly reduce this list of start states from which each component can be evaluated.

Algorithm 1 MAXEXP method for calculating upper bounds for parents in the role allocation space.

```
1: for all parent in search space do
2:   MAXEXP[parent] $\leftarrow 0$
3:   for all component $i$ corresponding to factors in the RMTDP from Section 4.1 do
4:     if component $i$ has a preceding component $j$ then
5:       Obtain start states, $states[i] \leftarrow endStates[j]$
6:       $states[i] \leftarrow$ removeIrrelevantFeatures($states[i]$) {discard features not present in $S_i$}
7:       Obtain corresponding observation histories at start $OHistories[i] \leftarrow endOHistories[j]$
8:       $OHistories[i] \leftarrow$ removeIrrelevantObservations($OHistories[i]$)
9:     else
10:      Obtain start states, $states[i]$
11:      Observation histories at start $OHistories[i] \leftarrow$ null
12:    $maxEval[i] \leftarrow 0$
13:    for all leaf-level policies $\pi$ under parent do
14:      $maxEval[i] \leftarrow \mathbf{max}(maxEval[i], \mathbf{max}_{s_i \in states[i], oh_i \in OHistories[i]} (Evaluate(RMTDP_i, s_i, oh_i, \pi)))$
15:  MAXEXP[parent] $\overset{+}{\leftarrow} maxEval[i]$
```

Similarly, the starting observation histories for a component are the observation histories on completing the preceding component (no observation history for the very first component). BDI plans do not normally refer to entire observation histories but rely only on key beliefs which are typically referred to in the pre-conditions of the component. Each starting observation history can be shortened to include only these relevant observations, thus obtaining a reduced list of starting observation sequences. Divergence of private observations is not problematic, e.g. will not cause agents to trigger different team plans. This is because as indicated earlier in Section 2.2, TOP interpreters guarantee coherence in key aspects of observation histories. For instance, as discussed earlier, TOP interpreter ensures coherence in key beliefs when initiating and terminating team plans in a TOP; thus avoiding such divergence of observation histories.

In order to compute the maximum value for a particular component, we evaluate all possible leaf-level policies within that component over all possible start states and observation histories and obtain the maximum (Algorithm 1:steps 13-14). During this evaluation, we store all the end states and ending observation histories so that they can be used in the evaluation of subsequent components. As shown in Figure 9, for the evaluation of **DoScouting** component for the parent node where there are two helicopters assigned to *Scouting Team* and four helos to *Transport Team*, the leaf-level policies correspond to all possible ways these helicopters could be assigned to the teams *SctTeamA*, *SctTeamB*, *SctTeamC* and *Transport Team*, e.g. one helo to *SctTeamB*, one helo to *SctTeamC* and four helos to *Transport Team*, or two helos to *SctTeamA* and four helos to *Transport Team*, etc. The role allocation tells the agents what role to take in the first step. The remainder of the role-taking policy is specified by the role replacement policy in the TOP infrastructure and role-execution policy is specified by the **DoScouting** component of the TOP.

To obtain the MaxEstimate for a parent node of the role allocation space, we simply sum up the maximum values obtained for each component (Algorithm 1:steps 15), e.g. the maximum values of each component (see right of each component in Figure 9) were summed to obtain the MaxEstimate $(84 + 3330 + 36 = 3420)$. As seen in Figure 8, third node from the left indeed has an upper bound of 3420.

The calculation of the MaxEstimate for a parent nodes should be much faster than evaluating the leaf nodes below it in most cases for two reasons. Firstly, parent nodes are evaluated component-wise. Thus, if multiple leaf-level policies within one component result in the same end state, we can remove duplicates to get the start states of the next component. Since each component only contains the state features relevant to it, the number of duplicates is greatly increased. Such duplication of the evaluation effort cannot be avoided for leaf nodes, where each policy is evaluated independently from start to finish. For instance, in the **DoScouting** component, the role allocation, *SctTeamA=1, SctTeamB=1, SctTeamC=0, TransportTeam=4* and the role allocation *SctTeamA=1, SctTeamB=0, SctTeamC=1, TransportTeam=4* will have end states in common after eliminating irrelevant features when the scout in *SctTeamB* for the former allocation and the scout in *SctTeamC* for the latter allocation fail. This is because through feature elimination (Algorithm 1:steps 6), the only state features retained for the **DoTransport** component are the scouted route and number of transports (some transports may have replaced failed scouts) as shown in Figure 9.

The second reason computation of MaxEstimates for parents is much faster is that the number of starting observation sequences will be much less than the number of ending observation histories of the preceding components. This is because not all the observations in the observation histories of a component are relevant to its succeeding components (Algorithm 1:steps 8). Thus, the function **removeIrrelevantObservations** reduces the number of starting observation histories from the observation histories of the preceding component.

We refer to this methodology of obtaining the MaxEstimates of each parent as MAXEXP. A variation of this, the maximum expected reward with no failures (NOFAIL), is obtained in a similar fashion except that we assume that the probability of any agent failing is 0. We are able to make such an assumption in

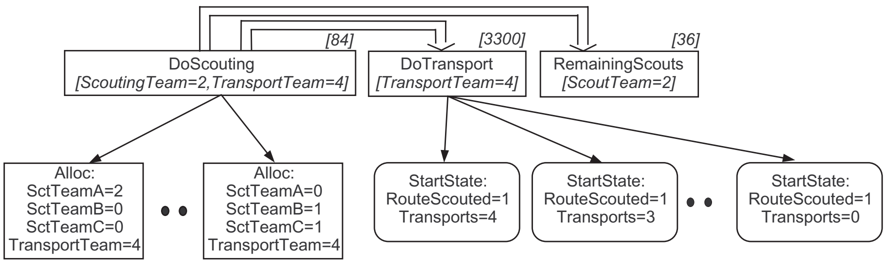


Fig. 9. Component-wise decomposition of a parent by exploiting TOP

evaluating the parent node, since we focus on obtaining upper bounds of parents, and not on obtaining their exact value. This will result in less branching and hence evaluation of each component will proceed much quicker. The NOFAIL heuristic only works if the evaluation of any policy without failures occurring is higher than the evaluation of the same policy with failures possible. This should normally be the case in most domains. The evaluation of the NOFAIL heuristics for the role allocation space for six helicopters is shown in square brackets in Figure 8.

6 Experimental Results

This section presents four sets of results in the context of the two domains introduced in Section 2.1, viz. mission rehearsal and RoboCupRescue [21]. First, we investigated empirically the speedups that result from using the TOP-congruent belief states ψ_i (belief-based evaluation) over observation history-based evaluation and from using the algorithm from Section 5 over a brute-force search. Here we focus on determining the best assignment of agents to roles; but assume a fixed TOP and TOP infrastructure. Second, we conducted experiments to investigate the benefits of considering uncertainty in determining role allocations. For this, we compared the allocations found by the RMTDP role allocation algorithm with (i) allocations which do not consider any kind of uncertainty, and (ii) allocations which do not consider observational uncertainty but consider action uncertainty. Third, we conducted experiments in both domains to determine the sensitivity of the results to changes in the model. Fourth, we compare the performance of allocations found by the RMTDP role allocation algorithm with allocations of human subjects in the more complex of our domains – RoboCupRescue simulations.

6.1 Results in Mission Rehearsal Domain

For the mission rehearsal domain, the TOP is the one discussed in Section 2.2. As can be seen in Figure 4(a), the organization hierarchy requires determining the number of agents to be allocated to the three scouting sub-teams and the remaining helos must be allocated to the transport sub-team. Different numbers

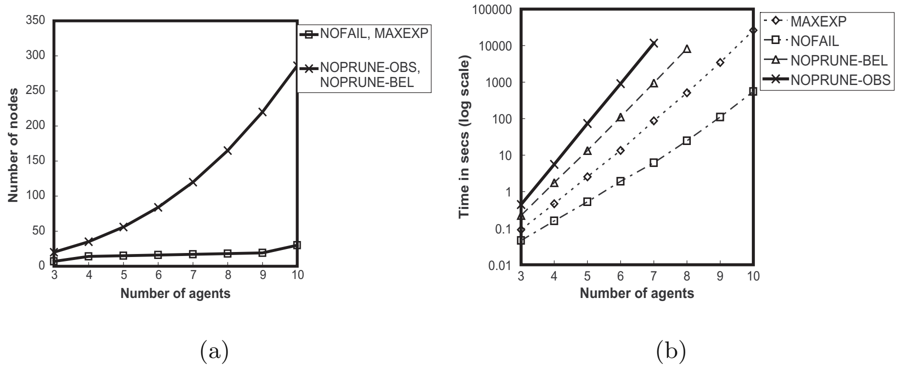


Fig. 10. Performance of role allocation space search in mission rehearsal domain, a) Number of nodes evaluated, b) Run-time in seconds on a log scale

of initial helicopters were attempted, varying from three to ten. The probability of failure of a scout at each time step on routes 1, 2 and 3 are 0.1, 0.15 and 0.2, respectively. The probability of a transport observing an alive scout on routes 1, 2 and 3 are 0.95, 0.94 and 0.93, respectively. False positives are not possible, i.e. a transport will not observe a scout as being alive if it has failed. The probability of a transport observing a scout failure on routes 1, 2 and 3 are 0.98, 0.97 and 0.96, respectively. Here too, false positives are not possible and hence a transport will not observe a failure unless it has actually taken place.

Figure 10 shows the results of comparing the different methods for searching the role allocation space. We show four methods. Each method adds new speedup techniques to the previous:

1. NOPRUNE-OBS: A brute force evaluation of every role allocation to determine the best. Here, each agent maintains its complete observation history and the evaluation algorithm in Equation 1 is used. For ten agents, the RMTDP is projected to have in the order of 10,000 reachable states and in the order of 100,000 observation histories per role allocation evaluated (thus the largest experiment in this category was limited to seven agents).
2. NOPRUNE-BEL: A brute force evaluation of every role allocation. The only difference between this method and NOPRUNE-OBS is the use of the belief-based evaluation algorithm (see Equation 2).
3. MAXEXP: The branch-and-bound search algorithm described in Section 5.2 that uses upper bounds of the evaluation of the parent nodes to find the best allocation. Evaluation of the parent and leaf nodes uses the belief-based evaluation.
4. NOFAIL: The modification to branch-and-bound heuristic mentioned in Section 5.2. In essence it is same as MAXEXP, except that the upper bounds are computed making the assumption that agents do not fail. This heuristic is correct in those domains where the total expected reward with failures is always less than if no failures were present and will give significant speedups

if agent failures is one of the primary sources of stochasticity. In this method, too, the evaluation of the parent and leaf nodes uses the belief-based evaluation. (Note that only upper bounds are computed using the no-failure assumption – no changes are assumed in the actual domains.)

In Figure 10(a), the Y-axis is the number of nodes in the role allocation space evaluated (includes leaf nodes as well as parent nodes), while in Figure 10(b) the Y-axis represents the runtime in seconds on a *logarithmic* scale. In both figures, we vary the number of agents on the X-axis. Experimental results in previous work using distributed POMDPs are often restricted to just two agents; by exploiting hybrid models, we are able to vary the number of agents from three to ten as shown in Figure 10(a). As clearly seen in Figure 10(a), because of pruning, significant reductions are obtained by MAXEXP and NOFAIL over NOPRUNE-BEL in terms of the numbers of nodes evaluated. This reduction grows quadratically to about 10-fold at ten agents.[1] NOPRUNE-OBS is identical to NOPRUNE-BEL in terms of number of nodes evaluated, since in both methods all the leaf-level policies are evaluated, only the method of evaluation differs. It is important to note that although NOFAIL and MAXEXP result in the same number of nodes being evaluated for this domains, this is not necessarily true always. In general, NOFAIL will evaluate at least as many nodes as MAXEXP since its estimate is at least as high as the MAXEXP estimate. However, the upper bounds are computed quicker for NOFAIL.

Figure 10(b) shows that the NOPRUNE-BEL method provides a significant speedup over NOPRUNE-OBS in actual run-time. For instance, there was a 12-fold speedup using NOPRUNE-BEL instead of NOPRUNE-OBS for the seven agent case (NOPRUNE-OBS could not be executed within a day for problem settings with greater than seven agents). This empirically demonstrates the computational savings possible using belief-based evaluation instead of observation history-based evaluation (see Section 4). For this reason, we use only belief-based evaluation for the MAXEXP and NOFAIL approaches and also for all the remaining experiments in this paper. MAXEXP heuristic results in a 16-fold speedup over NOPRUNE-BEL in the eight agent case.

The NOFAIL heuristic which is very quick to compute the upper bounds far outperforms the MAXEXP heuristic (47-fold speedup over MAXEXP for ten agents). Speedups of MAXEXP and NOFAIL continually increase with increasing number of agents. The speedup of the NOFAIL method over MAXEXP is so marked because, in this domain, ignoring failures results in much less branching.

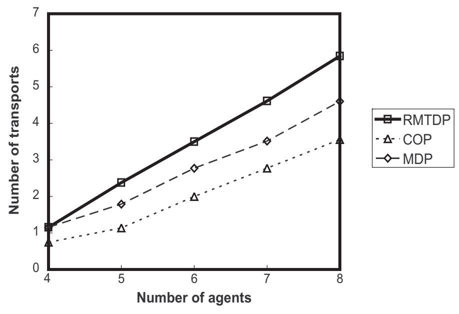
Next, we conducted experiments illustrating the importance of RMTDP's reasoning about action and observation uncertainties on role allocations. For this, we compared the allocations found by the RMTDP role allocation algorithm with allocations found using two different methods (see Figure 11):

[1] The number of nodes for NOPRUNE up to eight agents were obtained from experiments, the rest can be calculated using the formula $[m]^n/n! = (m+n-1)\cdot\ldots\cdot m/n!$, where m represents the number of heterogeneous role types and n is the number of homogeneous agents. $[m]^n = (m+n-1)\cdot\ldots\cdot m$ is referred to as a rising factorial.

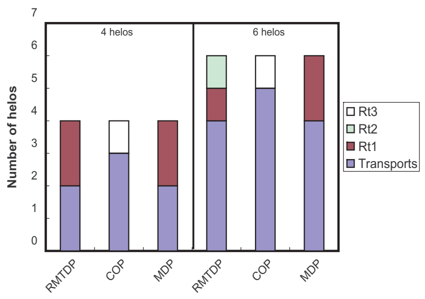
1. Role allocation via constraint optimization (COP [24, 23]) allocation approach: In the COP approach[2], leaf-level sub-teams from the organization hierarchy are treated as variables and the number of helicopters as the domain of each such variable (thus, the domain may be 1, 2, 3,..helicopters). The reward for allocating agents to sub-teams is expressed in terms of constraints:
   – Allocating a helicopter to scout a route was assigned a reward corresponding to the route's distance but ignoring the possibility of failure (i.e. ignoring transition probability). Allocating more helicopters to this subteam obtained proportionally higher reward.
   – Allocating a helicopter a transport role was assigned a large reward for transporting cargo to the destination. Allocating more helicopters to this subteam obtained proportionally higher reward.
   – Not allocating at least one scout role was assigned a reward of negative infinity
   – Exceeding the total number of agents was assigned a reward of negative infinity
2. RMTDP with complete observability: In this approach, we consider the transition probability, but ignore partial observability; achieved by assuming complete observability in the RMTDP. An MTDP with complete observability is equivalent to a Markov Decision Problem (MDP) [30] where the actions are joint actions. We, thus, refer to this allocation method as the MDP method.

Figure 11(a) shows a comparison of the RMTDP-based allocation with the MDP allocation and the COP allocation for increasing number of helicopters (X-axis). We compare using the expected number of transports that get to the destination (Y-axis) as the metric for comparison since this was the primary objective of this domain. As can be seen, considering both forms of uncertainty (RMTDP) performs better than just considering transition uncertainty (MDP) which in turn performs better than not considering uncertainty (COP). Figure 11(b) shows the actual allocations found by the three methods with four helicopters and with six helicopters. In the case of four helicopters (first three bars), RMTDP and MDP are identical, two helicopters scouting route 2 and two helicopters taking on transport role. The COP allocation however consists of one scout on route 3 and three transports. This allocation proves to be too myopic and results in fewer transports getting to the destination safely. In the case of six helicopters, COP chooses just one scout helicopter on route 3, the shortest route. The MDP approach results in two scouts both on route 1, which was longest route albeit the safest. The RMTDP approach, which also considers observational uncertainty choose to allocate the two scouts to route 1 and route 2, in order to take care of the cases where failures of scouts go undetected by the transports.

[2] Modi et al.'s work [24] focused on decentralized COP, but in this investigation our emphasis is on the resulting role allocation generated by the COP, and not on the decentralization per se.

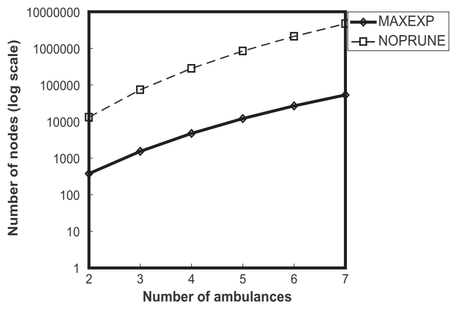


(a)

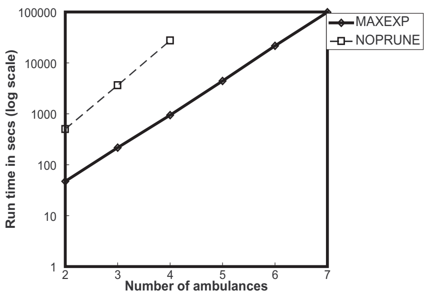


(b)

Fig. 11. a) Comparison of performance of different allocation methods, b) Allocations found using different allocation methods



(a)



(b)

Fig. 12. Performance of role allocation space search in RoboCupRescue, a) Number of nodes evaluated on a log scale, and b) Run-time in seconds on a log scale

6.2 Results in RoboCupRescue Domain

Speedups in RoboCupRescue Domain In our next set of experiments, we highlight the computational savings obtained in the RoboCupRescue domain. The scenario for this experiment consisted of two fires at different locations in the city. Each of these fires has a different initially unknown number of civilians in it, however the total number of civilians and the distribution from which the locations of the civilians is chosen is known ahead of time. For this experiment, we fix the number of civilians at five and set the distribution used to choose the civilians' locations to be uniform. The number of fire engines is set at five, located in three different fire stations as described in Section 2.1 and vary the number of ambulances, all co-located at an ambulance center, from two to seven. The reason we chose to change only the number of ambulances is because small number of fire engines are unable to extinguish fires, changing the problem completely.

The goal is to determine which fire engines to allocate to which fire and once information about civilians is transmitted, how many ambulances to send to each fire location.

Figure 12 highlights the savings in terms of the number of nodes evaluated and the actual runtime as we increase the number of agents. We show results only from NOPRUNE-BEL and MAXEXP. NOPRUNE-OBS could not be run because of slowness. Here the NOFAIL heuristic is identical to MAXEXP since agents cannot fail in this scenario. The RMTDP in this case had about 30,000 reachable states.

In both Figures 12(a) and 12(b), we increase the number of ambulances along the X-axis. In Figure 12(a), we show the number of nodes evaluated (parent nodes + leaf nodes)[3] on a logarithmic scale. As can be seen, the MAXEXP method results in about a 89-fold decrease in the number of nodes evaluated when compared to NOPRUNE-BEL for seven ambulances, and this decrease becomes more pronounced as the number of ambulances is increased. Figure 12(b) shows the time in seconds on a logarithmic scale on the Y-axis and compares the run-times of the MAXEXP and NOPRUNE-BEL methods for finding the best role allocation. The NOPRUNE-BEL method could not find the best allocation within a day when the number of ambulances was increased beyond four. For four ambulances (and five fire engines), MAXEXP resulted in about a 29-fold speedup over NOPRUNE-BEL.

Allocation in RoboCupRescue. Our next set of experiments shows the practical utility of our role allocation analysis in complex domains. We are able to show significant performance improvements in the actual RoboCupRescue domain using the role allocations generated by our analysis. First, we construct an RMTDP for the rescue scenario, described in Section 2.1, by taking guidance from the TOP and the underlying domain (as described in Section 4.1). We then use the MAXEXP heuristic to determine the best role allocation. We compared the RMTDP allocation with the allocations chosen by human subjects. Our goal in comparing RMTDP allocations with human subjects was mainly to show that RMTDP is capable at performing at or near human expert levels for this domain. In addition, in order to determine that reasoning about uncertainty actually impacts the allocations, we compared the RMTDP allocations with allocations determined by two additional allocation methods:

1. RescueISI: Allocations used by the our RoboCupRescue agents that were entered in the RoboCupRescue competitions of 2001 [25] (RescueISI), where they finished in third place. These agents used local reasoning for their decision making, ignoring transitional as well and observational uncertainty.

[3] The number of nodes evaluated using NOPRUNE-BEL can be computed as $(f_1+1)\cdot(f_2+1)\cdot(f_3+1)\cdot(a+1)^{c+1}$, where f_1, f_2 and f_3 are the number of fire engines are station 1, 2 and 3, respectively, a is the number of ambulances and c is the number of civilians. Each node provides a complete conditional role allocation, assuming different numbers of civilians at each fire station.

2. RMTDP with complete observability: As discussed earlier, complete observability in RMTDP leads to an MDP, and we refer to this method as the MDP method.

Note that these comparisons were performed using the RoboCupRescue simulator with multiple runs to deal with stochasticity[4]. The scenario is as described in Section 6.2. We fix the number of fire engines, ambulances and civilians at five each. For this experiment, we consider two settings, where the location of civilians is drawn from:

– Uniform distribution – 25% of the cases have four civilians at fire 1 and one civilian at fire 2, 25% with three civilians at fire 1 and two at fire 2, 25% with two civilians at fire 1 and three at fire 2 and the remaining 25% with one civilian at fire 1 and four civilians at fire 2. The speedup results of Section 6.2 were obtained using this distribution.
– Skewed distribution – 80% of the cases have four civilians at fire 1 and one civilian at fire 2 and the remaining 20% with one civilian at fire 1 and four civilians at fire 2.

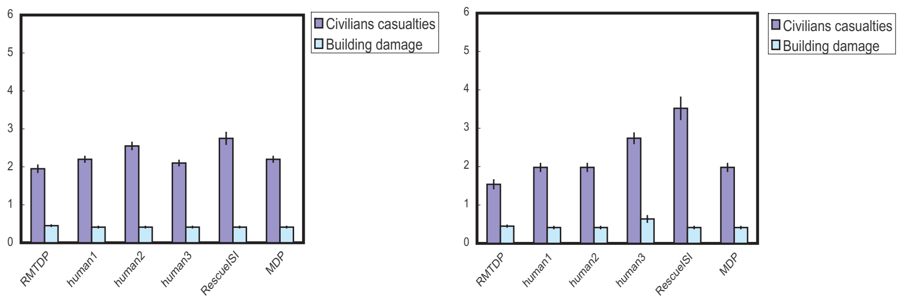
Note that we do not consider the case where all civilians are located at the same fire as the optimal ambulance allocation is simply to assign all ambulances to the fire where the civilians are located. A skewed distribution was chosen to highlight the cases where it becomes difficult for humans to reason about what allocation to choose.

The three human subjects used in this experiment were researchers at USC. All three were familiar with RoboCupRescue. They were given time to study the setup and were not given any time limit to provide their allocations. Each subject was told that the allocations were going to be judged first on the basis of the number of civilian lives lost and next on the damage sustained due to fire. These are exactly the criteria used in RoboCupRescue [21].

We then compared "RMTDP" allocation with those of the human subjects in the RoboCupRescue simulator and with RescueISI and MDP. In Figure 13, we compared the performance of the allocations on the basis of the number of civilians who died and the average damage to the two buildings (lower values are better for both criteria). These two criteria are the main two criteria used in RoboCupRescue [21]. The values shown in Figure 13 were obtained by averaging forty simulator runs for the uniform distribution and twenty runs for the skewed distribution for each allocation. The average values were plotted to account for the stochasticity in the domain. Error bars are provided to show the standard error for each allocation method.

As can be seen in Figure 13(a), the RMTDP allocation did better than the other five allocations in terms of a lower number of civilians dead (although

[4] For the mission rehearsal domain, we could not run on the actual mission rehearsal simulator since that simulator is not public domain and no longer accessible, and hence the difference in how we tested role allocations in the mission rehearsal and the RoboCupRescue domains.



(a) (b)

Fig. 13. Comparison of performance in RoboCupRescue, a) uniform, and b) skewed

human3 was quite close). For example, averaging forty runs, the RMTDP allocation resulted in 1.95 civilian deaths while human2's allocation resulted in 2.55 civilian deaths. In terms of the average building damage, the six allocations were almost indifferentiable, with the humans actually performing marginally better. Using the skewed distribution, the difference between the allocations was much more perceptible (see Figure 13(b)). In particular, we notice how the RMTDP allocation does much better than the humans in terms of the number of civilians dead. Here, human3 did particularly badly because of a bad allocation for fire engines. This resulted in more damage to the buildings and consequently to the number of civilians dead.

Comparing RMTDP with RescueISI and the MDP approach showed that reasoning about transitional uncertainty (MDP) does better than a static reactive allocation method (RescueISI) but not as well as reasoning about both transitional and observational uncertainty. In the uniform distribution case, we found that RMTDP does better than both MDP and RescueISI, with the MDP method performing better than RescueISI. In the skewed distribution case, the improvement in allocations using RMTDP is greater. Averaging twenty simulation runs, RMTDP allocations resulted in 1.54 civilians deaths while MDP resulted in 1.98 and RescueISI in 3.52. The allocation method used by RescueISI often resulted in one of the fires being allocated too few fire engines. The allocations determined by the MDP approach turned out to be the same as human1.

A two-tailed t-test was performed in order to test the statistical significance of the means for the allocations in Figure 13. The means of number of civilians dead for the RMTDP allocation and the human allocations were found to be statistically different (confidence $> 96\%$) for both the uniform as well as the skewed distributions. The difference in the fire damage was not statistically significant in the uniform case, however, the difference between the RMTDP allocation and human3 for fire damage was statistically significant ($> 96\%$) in the skewed case.

These experiments show that the allocations found by the RMTDP role allocation algorithm performs significantly better than allocations chosen by human subjects and RescueISI and MDP in most cases (and does not do significantly worse in any case). In particular when the distribution of civilians is not uniform, it is more difficult for humans to come up with an allocation and the difference between human allocations and the RMTDP allocation becomes more significant. From this we can conclude that the RMTDP allocation performs at near-human expertise.

7 Related Work

There are three related areas of research that we wish to highlight. First, there has been a considerable amount of work done in the field of multiagent teamwork (Section 7.1). The second related area of research is the use of decision theoretic models, in particular distributed POMDPs (Section 7.2). Finally, in Section 7.3, the related work in role allocation and reallocation in multiagent teams is described.

7.1 BDI-Based Teamwork

Several formal teamwork theories such as *Joint Intentions* [6], *SharedPlans* [14] were proposed that tried to capture the essence of multiagent teamwork in the logic of Beliefs-Desires-Intentions. Next, practical models of teamwork such as COLLAGEN [32], GRATE* [19], STEAM [37] built on these teamwork theories [6, 14] and attempted to capture the aspects of teamwork that were reusable across domains. In addition, to complement the practical teamwork models, the team-oriented programming approach [31, 39] was introduced to allow large number of agents to be programmed as teams. This approach was then expanded on and applied to a variety of domains [31, 44, 7]. Other approaches for building practical multiagent systems [36, 9], while not explicitly based on team-oriented programming, could be considered in the same family.

The research reported in this paper complements this research on teamwork by introducing hybrid BDI-POMDP models that exploit the synergy between BDI and POMDP approaches. In particular, TOP and teamwork models have traditionally not addressed uncertainty and cost. Our hybrid model provides this capability, and we have illustrated the benefits of this reasoning via detailed experiments.

While this paper uses team-oriented programming [38, 7, 39] as an example BDI approach, it is relevant to other similar techniques of modeling and tasking collectives of agents, such as Decker and Lesser's [9] TAEMS approach. In particular, the TAEMS language provides an abstraction for tasking collaborative groups of agents similar to TOP, while the GPGP infrastructure used in executing TAEMS-based tasks is analogous to the "TOP interpreter" infrastructure shown in Figure 1. While Lesser et al. have explored the use of distributed MDPs in analyses of GPGP coordination [42], they have not exploited the use of TAEMS structures in decomposition or abstraction for searching optimal policies in distributed MDPs, as suggested in this paper. Thus, this paper complements Lesser et al.'s work in illustrating a significant avenue for further efficiency improvements in such analyses.

7.2 Distributed POMDP Models

Three different approaches have been used to solve distributed POMDPs. One approach that is typically taken is to make simplifying assumptions about the domain. For instance, in Guestrin et al. [15], it is assumed that each agent can completely observe the world state. In addition, it is assumed that the reward function (and transition function) for the team can be expressed as the sum (product) of the reward (transition) functions of the agents in the team. Becker et al. [2] assume that the domain is factored such that each agent has a completely observable local state and also that the domain is transition-independent (one agent cannot affect another agent's local state).

The second approach taken is to simplify the nature of the policies considered for each of the agents. For example, Chadès et al. [5] restrict the agent policies to be memoryless (reactive) policies, thereby simplifying the problem to solving multiple MDPs. Peshkin et al. [29] take a different approach by using gradient descent search to find local optimum finite-controllers with bounded memory. Nair et al. [26, 27] present an algorithm for finding a locally optimal policy from a space of unrestricted finite-horizon policies. The third approach, taken by Hansen et al. [16], involves trying to determine the globally optimal solution without making any simplifying assumptions about the domain. In this approach, they attempt to prune the space of possible complete policies by eliminating dominated policies. Although a brave frontal assault on the problem, this method is expected to face significant difficulties in scaling up due to the fundamental complexity of obtaining a globally optimal solution.

The key difference with our work is that our research is focused on hybrid systems where we leverage the advantages of BDI team plans, which are used in practical systems, and distributed POMDPs that quantitatively reason about uncertainty and cost. In particular, we use TOPs to specify large-scale team plans in complex domains and use RMTDPs for finding the best role allocation for these teams.

7.3 Role Allocation and Reallocation

There are several different approaches to the problem of role allocation and reallocation. For example, Tidhar et al. [40] and Tambe et al. [38] performed role allocation based on matching of capabilities, while Hunsberger and Grosz [18] proposed the use of combinatorial auctions to decide on how roles should be assigned. Modi et al. [24] showed how role allocation can be modeled as a distributed constraint optimization problem and applied it to the problem of tracking multiple moving targets using distributed sensors. Shehory and Kraus [34] suggested the use of coalition formation algorithms for deciding quickly which agent took on which role. Fatima and Wooldridge [11] use auctions to decide on task allocation. It is important to note that these competing techniques are not free of the problem of how to model the problem, even though they do not have to model transition probabilities. Other approaches to reforming a team are reconfiguration methods due to Dunin-Keplicz and Verbrugge [10], self-adapting organizations by Horling and Lesser [17] and dynamic re-organizing groups [1]. Scerri et al. [33] present a role (re)allocation algorithm that allows autonomy of role reallocation to shift between a human supervisor and the agents.

The key difference with all this prior work is our use of stochastic models (RMTDPs) to evaluate allocations: this enables us to compute the benefits of role allocation, taking into account uncertainty and costs of reallocation upon failure. For example, in the mission rehearsal domain, if uncertainties were not considered, just one scout would have been allocated, leading to costly future reallocations or even in mission failure. Instead, with lookahead, depending on the probability of failure, multiple scouts were sent out on one or more routes, resulting in fewer future reallocations and higher expected reward.

8 Conclusion

While the BDI [38, 9, 39] approach to agent teamwork has provided successful applications, tools and techniques that provide quantitative analyses of team coordination and other team behaviors under uncertainty are lacking. The emerging field of distributed POMDPs [3, 4, 30, 43, 29, 16] provides a rich framework for modeling uncertainties and utilities in complex multiagent domains. However, as shown by Bernstein et al. [3], the problem of deriving the optimal policy is computationally intractable.

In order to deal with this issue of intractability in distributed POMDPs, this paper presented a principled way to combine the two dominant paradigms for building multiagent teams, namely the BDI approach and distributed POMDPs. In this hybrid BDI-POMDP approach, BDI team plans are exploited to improve distributed POMDP tractability and distributed POMDP-based analysis improves BDI team plan performance. Thus, this approach leverages the benefits of both the BDI and POMDP approaches to analyze and improve key coordination decisions within BDI-based team plans using POMDP-based methods. In order to demonstrate these analysis methods, we concentrated on role allocation – a fundamental aspect of agent teamwork. First, we introduced RMTDP, a distributed POMDP based framework, for analysis of role allocation. Second, this paper presented an RMTDP-based methodology for optimizing key coordination decisions within a BDI team plan for a given domain. Concretely, the paper described a methodology for finding the best role allocation for a fixed team plan. Given the combinatorially many role allocations, we introduced methods to exploit task decompositions among sub-teams to significantly prune the search space of role allocations.

We constructed RMTDPs for two domains – RoboCupRescue and mission rehearsal simulation – and determined the best role allocation in these domains. Furthermore, we illustrated significant speedups in RMTDP policy search due to the techniques introduced in this paper. Detailed experiments revealed the advantages of our approach over state-of-the-art role allocation approaches that do not reason with uncertainty. In the RoboCupRescue domain, we showed that the role allocation technique presented in this paper is capable of performing at human expert levels by comparing with the allocations chosen by humans in the actual RoboCupRescue simulation environment.

References

1. S. Barber and C. Martin. Dynamic reorganization of decision-making groups. In *Proceedings of the Fifth International Conference on Autonomous Agents (Agents-01)*, pages 513–520, 2001.
2. R. Becker, S. Zilberstein, V. Lesser, and C. V. Goldman. Transition-independent decentralized Markov decision processes. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS-03)*, pages 41–48, 2003.
3. D. S. Bernstein, S. Zilberstein, and N. Immerman. The complexity of decentralized control of MDPs. In *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence(UAI-00)*, pages 32–37, 2000.
4. C. Boutilier. Planning, learning & coordination in multiagent decision processes. In *Proceedings of the Sixth Conference on Theoretical Aspects of Rationality and Knowledge (TARK-96)*, pages 195–210, 1996.
5. I. Chadès, B. Scherrer, and F. Charpillet. A heuristic approach for solving decentralized-pomdp: Assessment on the pursuit problem. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC-02)*, pages 57–62, 2002.
6. P. R. Cohen and H. J. Levesque. Teamwork. *Nous*, 25(4):487–512, 1991.
7. J. L. T. da Silva and Y. Demazeau. Vowels co-ordination model. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002)*, pages 1129–1136, 2002.
8. T. Dean and S. H. Lin. Decomposition techniques for planning in stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1121–1129, 1995.
9. K. Decker and V. Lesser. Quantitative modeling of complex computational task environments. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 217–224, 1993.
10. B. Dunin-Keplicz and R. Verbrugge. A reconfiguration algorithm for distributed problem solving. *Engineering Simulation*, 18:227–246, 2001.
11. S. S. Fatima and M. Wooldridge. Adaptive task and resource allocation in multi-agent systems. In *Proceedings of the Fifth International Conference on Autonomous Agents (Agents-01)*, pages 537–544, May 2001.
12. M. P. Georgeff and A. L. Lansky. Procedural knowledge. *Proceedings of the IEEE special issue on knowledge representation*, 74:1383–1398, 1986.
13. B. Grosz, L. Hunsberger, and S. Kraus. Planning and acting together. *AI Magazine*, 20(4):23–34, 1999.
14. B. Grosz and S. Kraus. Collaborative plans for complex group action. *Artificial Intelligence*, 86(2):269–357, 1996.
15. C. Guestrin, S. Venkataraman, and D. Koller. Context specific multiagent coordination and planning with factored MDPs. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-02)*, pages 253–259, 2002.
16. E. A. Hansen, D. S. Bernstein, and S. Zilberstein. Dynamic programming for partially observable stochastic games. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, pages 709–715, 2004.

17. B. Horling, B. Benyo, and V. Lesser. Using self-diagnosis to adapt organizational structures. In *Proceedings of the Fifth International Conference on Autonomous Agents (Agents-01)*, pages 529–536, 2001.
18. L. Hunsberger and B. Grosz. A combinatorial auction for collaborative planning. In *Proceedings of the Fourth International Conference on Multiagent Systems (ICMAS-2000)*, pages 151–158, 2000.
19. N. Jennings. Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. *Artificial Intelligence*, 75(2):195–240, 1995.
20. L. Kaelbling, M. Littman, and A. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(2):99–134, 1998.
21. H. Kitano, S. Tadokoro, I. Noda, H. Matsubara, T. Takahashi, A. Shinjoh, and S. Shimada. RoboCup-Rescue: Search and rescue for large scale disasters as a domain for multiagent research. In *Proceedings of IEEE Conference on Systems, Men, and Cybernetics (SMC-99)*, pages 739–743, 1999.
22. H. J. Levesque, P. R. Cohen, and J. Nunes. On acting together. In *Proceedings of the National Conference on Artificial Intelligence*, pages 94–99. Menlo Park, Calif.: AAAI press, 1990.
23. R. T. Mailler and V. Lesser. Solving distributed constraint optimization problems using cooperative mediation.
24. P. J. Modi, W.-M. Shen, M. Tambe, and M. Yokoo. An asynchronous complete method for distributed constraint optimization. In *Proceedings of the Second International Joint Conference on Agents and Multiagent Systems (AAMAS-03)*, pages 161–168, 2003.
25. R. Nair, T. Ito, M. Tambe, and S. Marsella. Task allocation in the rescue simulation domain. In *RoboCup 2001: Robot Soccer World Cup V*, volume 2377 of *Lecture Notes in Computer Science*, pages 751–754. Springer-Verlag, Heidelberg, Germany, 2002.
26. R. Nair, D. Pynadath, M. Yokoo, M. Tambe, and S. Marsella. Taming decentralized POMDPs: Towards efficient policy computation for multiagent settings. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 705–711, 2003.
27. R. Nair, M. Roth, M. Yokoo, and M. Tambe. Communication for improving policy computation in distributed pomdps. In *Proceedings of the Third International Joint Conference on Agents and Multiagent Systems (AAMAS-04)*, pages 1098–1105, 2004.
28. R. Nair, M. Tambe, and S. Marsella. Role allocation and reallocation in multiagent teams: Towards a practical analysis. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS-03)*, pages 552–559, 2003.
29. L. Peshkin, N. Meuleau, K.-E. Kim, and L. Kaelbling. Learning to cooperate via policy search. In *Proceedings of the Sixteenth Conference in Uncertainty in Artificial Intelligence (UAI-00)*, pages 489–496, 2000.
30. D. V. Pynadath and M. Tambe. The communicative multiagent team decision problem: Analyzing teamwork theories and models. *Journal of Artificial Intelligence Research*, 16:389–423, 2002.
31. D. V. Pynadath and M. Tambe. Automated teamwork among heterogeneous software agents and humans. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 7:71–100, 2003.
32. C. Rich and C. Sidner. COLLAGEN: When agents collaborate with people. In *Proceedings of the First International Conference on Autonomous Agents (Agents-97)*, pages 284–291, 1997.

33. P. Scerri, L. Johnson, D. Pynadath, P Rosenbloom, M. Si, N. Schurr, and M. Tambe. A prototype infrastructure for distributed robot, agent, person teams. In *Proceedings of the Second International Joint Conference on Agents and Multiagent Systems (AAMAS-03)*, pages 433–440, 2003.
34. O. Shehory and S. Kraus. Methods for task allocation via agent coalition formation. *Artificial Intelligence*, 101(1-2):165–200, 1998.
35. E. J. Sondik. The optimal control of partially observable Markov processes. *Ph.D. Thesis, Stanford*, 1971.
36. P. Stone and M. Veloso. Task decomposition, dynamic role assignment, and low-bandwidth communication for real-time strategic teamwork. *Artificial Intelligence*, 110(2):241–273, 1999.
37. M. Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, 7:83–124, 1997.
38. M. Tambe, D. Pynadath, and N. Chauvat. Building dynamic agent organizations in cyberspace. *IEEE Internet Computing*, 4(2):65–73, 2000.
39. G. Tidhar. Team-oriented programming: Social structures. Technical Report 47, Australian Artificial Intelligence Institute, 1993.
40. G. Tidhar, A. Rao, and E. Sonenberg. Guided team selection. In *Proceedings of the Second International Conference on Multi-agent Systems (ICMAS-96)*, pages 369–376, 1996.
41. M. Wooldridge. *An Introduction to Multiagent Systems*. John Wiley & Sons, 2002.
42. P. Xuan and V. Lesser. Multi-agent policies: from centralized ones to decentralized ones. In *Proceedings of the First International Joint Conference on Agents and Multiagent Systems (AAMAS-02)*, pages 1098–1105, 2002.
43. P. Xuan, V. Lesser, and S. Zilberstein. Communication decisions in multiagent cooperation. In *Proceedings of the Fifth International Conference on Autonomous Agents (Agents-01)*, pages 616–623, 2001.
44. J. Yen, J. Yin, T. R. Ioerger, M. S. Miller, D. Xu, and R. A. Volz. Cast: Collaborative agents for simulating teamwork. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 1135–1144, 2001.

Agents – The Challenge of Relevance to the IT Mainstream

David Kinny

Agentis Software
www.agentissoftware.com

Abstract. In the 10 years since the first ATAL workshop was held, Agent and Multi-Agent Systems have been a spectacular growth area of research in Computer Science. The field has certainly developed in terms of indicators such as number of publications, conferences and workshops. A view now expressed by many in the agent research community, and others, is that agents represent the 'next big thing' in software development and are poised to supplant object-oriented approaches. But is there any realistic prospect of this happening? Is the state-of-the-art in agents and the focus of agent research really relevant to enterprise computing? What might enterprise-ready agent technology look like? What factors would drive enterprises to invest in such solutions? This talk will attempt to analyze underlying issues and offer some answers to these questions.

1 Introduction

In the last decade agent researchers have developed many agent specification and programming languages, variously based on plans, rules, logics or constraints, or by extending conventional OO languages such as Java. As well as new languages, a handful of programming frameworks have been developed, typically in Java, as have hundreds of approaches, architectures, logics, techniques & algorithms, thousands of small-scale research applications, scores of methodologies, models and diagrams, and even a few agent-specific standards.

Clearly the agent research community has developed momentum and grown dramatically. The claim is now often made that agents are the most important recent development in Computer Science, and are poised to supplant existing ways to build software systems, i.e., objects. But the evidence to support this claim does not yet seem to be there. Of the small number of agent programming languages for which robust implementations have been developed, very few are used commercially. The development of agents in conventional OO languages predominates, even in research labs, and use of agents in commerce, industry, administration and defence is still a very limited activity, conducted only by specialists. There is enormous diversity within the agent community, to the point that two agent approaches or technologies may have almost nothing in common. So it seems reasonable to pose the question: are the agent paradigm and its various technologies really relevant to mainstream IT, and ready for prime time?

In considering this question it is instructive to look for possible parallels from Computer Science history. There have certainly been many other paradigms that promised to transform software development but weren't ever adopted by the IT mainstream. Examples include Lisp, logic programming languages such as Prolog, Parlog and Mercury; expert and rule-based systems; active databases; functional programming languages such as Scheme, ML, Miranda, and Haskell; and multi-paradigm languages, such as Oz and its successor Mozart.

What lies behind their failure to impact mainstream software development? The reasons are surely many and complex, but include that these languages and their technologies

– were too great a paradigm shift,
– were insufficiently mature or complete,
– were over-hyped and came to be viewed as failing to deliver on their promises,
– were useful only for niche applications, and
– needed "rocket scientists" to exploit their power.

Perhaps most importantly, they

– didn't really fit into the enterprise,
– were unable to integrate well with and leverage existing software assets, and
– didn't address real enterprise needs or pain.

It is certainly not a question of the the "quality" of these software paradigms, which have been eminently successful from a research and teaching perspective, and have in some cases achieved a level of niche success in specialized enterprises. So what does it take to make a real impact on enterprise computing, and do agent approaches have any better prospect of mainstream adoption? If so, which agent approaches? And how can they avoid repeating the failures of these other paradigms?

Today's enterprise IT standards are the outcome of the 30+ year maturation of OO languages and transaction processing. Evolution rather than revolution is the nature of the game, and lead times for the adoption of advanced technologies can be substantial. To accelerate this process, new software paradigms must adequately address enterprise needs and deliver substantial benefits that cannot be achieved by conventional approaches.

Agentis believes that the agent paradigm has the potential to be widely adopted, and to deliver significant benefits to enterprise computing. It is clear, however, that most agent technologies do not yet meet the needs of the enterprise, and that there is little understanding amongst agent practitioners as to what these benefits are. The challenge for those in the agent community who focus on bringing agents into the mainstream is to develop and refine agent technologies that are an effective solution to enterprise needs, and to demonstrate clearly the benefits of their adoption. This talk will addresses these issues in more detail, proposing an approach that fuses the most valuable aspects of agent technology with the state-of-the-art in enterprise computing.

2 What Are Agents and MAS Really Good for?

Agent and MAS techniques have proven to be valuable for developing applications such as

- embedded and distributed monitoring and control systems,
- cooperative teams of robots or softbots,
- automated negotiation systems and marketplaces,
- simulation systems, animation systems, and games,
- certain types of optimisation systems, and
- intelligent assistant software.

So what benefits can agents offer the world of enterprise computing? To answer this question adequately one must consider which agent techniques and technologies are ready for mainstream use, how they might be used, what are the real needs of enterprise IT, and what, the beyond technologies themselves, are the prerequisites for their successful application?

There is certainly a real diversity of agent technologies, and of corresponding notions of the essence of the agent paradigm. Historically, the notion of agents as software with mental states and attitudes has been a key thread running through agent research, but this idea has been less influential in shaping the actual agent technologies in use today. To some, autonomy is a key aspect, leading to the idea of agents as some sort of heavyweight software component with exclusive access to computational resources. To others agents are active objects, perhaps elements of a swarm, and agent systems are taken to produce useful behaviours by emergence rather than top-down design. Perhaps the most widespread view is that the key aspect of agency is sociality, manifest as asynchronous communication in an abstract communication language, and the possession of knowledge about other agents, and perhaps also of social norms. Without dismissing the value of these ideas, we subscribe to an alternative view: that the essence of agency is the ability to reason about how to act in the pursuit of goals, i.e., agents as the embodiment of suitable representations of goals and behaviour, and computational mechanisms for decision making.

A prerequisite for the widespread adoption of a new software development paradigm is simplicity and accessibility of its core concepts and techniques. In the case of agents, there is no real consensus within the community on what these should be. Possibilities include the use of

- mentalistic concepts, e.g., beliefs, goals, intentions, norms, etc.,
- speech act based communication languages,
- specific component or system architectures,
- specific computational models for concurrency,
- algorithms and interaction mechanisms, e.g. auctions,
- graphical, process based programming languages,
- knowledge representation and reasoning techniques, and
- agent analysis and design methodologies, and their models.

At one extreme, the agent paradigm could be seen as an entirely new and dramatically different approach to software development, with new languages, architectures, components, and techniques for representation, decision making and interaction. The chances of mainstream adoption of such a large shift in practices are slim indeed, and in any case, such a coherent and comprehensive agent paradigm does not yet exist, and hence lacks experienced practitioners, teachers, and students who can carry it forth into the world of enterprise IT.

At the other extreme, it could be little more than conceptual and methodological contributions that extend existing approaches to the analysis, design and implementation of conventional OO systems: a few key patterns and techniques, along with suitably extended models and supporting tools. If it took that form its adoption might well be straightforward, if the benefits were apparent to mainstream practitioners, but as a paradigm it would struggle to distinguish itself and more than likely simply be absorbed into OO.

Agentis has identified and pursued a strategy for the development of agent technology that tries to find a balance between these extremes. Its essence is achieving a fusion of the most valuable aspects of agents and modern enterprise technologies, streamlining core agent concepts to facilitate their understanding, and developing a complete package of development environment, tools and platforms, along with methodological support for the entire development cycle. This approach can deliver the benefits of agent technology to the enterprise without demanding a revolutionary change in its practices. To understand these benefits we must focus on the needs and wants of the enterprise.

3 What Does the Typical Enterprise Want from IT

The driver for business investment in new software systems is to increase competitive advantage or address nagging problems that negatively impact profitability. Corporate objectives are to slash the total cost of ownership of IT systems and improve return on investment. System reliability, availability, and scalability are key requirements, as is the ability to easily and rapidly revise and extend functionality. Current wisdom says this can only be achieved by *being adaptive*, which means:

– achieving rapid application development and revision cycles,
– maintaining visibility of business process and logic in applications,
– continuous application evolution with minimal downtime, and
– the ability to change business logic post system deployment.

Enterprise IT managers seek to control project risk, employing commodity software developers to produce easily maintained, future-proof systems that conform to industry standards and adopt best practices, such as use of Model-Driven Architectures. A current trend is to trade heavyweight development processes for agile development, building systems incrementally, designing as you go. This approach demands first class support for team development.

Software developers expect to use an Integrated Development Environment (IDE) that provides graphical views of application structure and content, and powerful tools to automate application generation, particularly Graphical User Interface (GUI), data access and plumbing elements, which typically account for about 60% of the overall application development effort. Support for easy refactoring and continuous integration is essential, as are methodologies to guide the entire development process, and tools to simplify testing and deployment into the production environment. Resistance to approaches that are dramatically different from what developers already know can be significant.

Typical enterprise application development starts with requirements that are rarely complete or accurate. Business logic is subject to ongoing evolution during and beyond development. Applications are layered upon databases and/or transactional data services and message buses. GUI's are typically browser based. Applications must support a high volume of users, possibly globally distributed, with significant sustained or peak transaction rates; to address these needs, they must often be distributed over redundant servers.

These characteristics and requirements, and the need for adaptivity, lead naturally to an enterprise application architecture stack consisting of four layers:

1. A technology layer providing operating systems, databases, message buses, and application server technologies such as J2EE and .NET,
2. An application layer containing conventional packaged software applications such as offerings by SAP, Seibel, and PeopleSoft,
3. A business service layer which exploits capabilities of the application layer to provide services associated with different elements of the enterprise, such as sales, production, delivery, employee and customer management,
4. A business process layer that orchestrates and controls the invocation of business services, and delivers them to internal and external customers.

Within such a layered architecture, agents can serve as the ideal enabling technology for the business service and process layers.

4 Delivering Real Benefits to the Enterprise

Enterprises don't want agent (or other) technology, they want adaptive applications and solutions that deliver real business benefits. Our experience has been that a fusion of agent and enterprise technologies can deliver four such benefits:

– significant increases in developer productivity and reduction in time, cost and risk of application development projects,
– an adaptive development process that can deal effectively with change and refinement of business requirements prior to application deployment,
– adaptive applications that are capable of operating robustly in the face of unplanned exceptions and changes in the operating environment, and
– an ability for business users to change the application logic without a need for IT involvement or redeployment of the application.

These are compelling advantages over conventional software development approaches. Agentis delivers these advantages with a mature, enterprise-ready agent development environment and runtime agent server platform that aligns with industry software standards. From an agent technology perspective, its key features are as follows.

– The business model is the application, and business analysts can participate directly in application development, refinement and maintenance.
– Business processes and logic are expressed as services, goals and plans defined in an expressive but conceptually simple visual language.
– Agents and their services are assembled and defined in a graphical modelling environment and automatically built and deployed, so that business requirements and application functionality are kept in sync.
– By exploiting mature application server technologies such as J2EE, deployed applications are highly reliable, scalable and performant.
– At runtime, agents deliver services by dynamically assembling appropriate end-to-end processes, given the specific requirements and available resources, and reasoning about how to deal with changing environments and exceptions.
– The entire application development process is supported by a comprehensive methodology.

5 Research Challenges for the ProMAS Community

Agentis has pursued one particular approach to bringing agent technology into the mainstream, based on simplifying and streamlining BDI agent technology and fusing it with the state-of-the-art in enterprise computing. But this is only a first step in bringing the full power of agents into the IT mainstream, and the ProMAS community could contribute much to the process were it so inclined. We conclude then with a list of research challenges for this community.

– To design an accessible general purpose agent programming language, based on a simple conceptual model, that is not too great a leap from existing OO languages such as Java, and which better supports agent structuring, code reuse, integration, and commercial data manipulation.
– To develop corresponding AO models and a methodology that effectively guides and supports the development of individual agents as well as MAS.
– To develop techniques for flexible structuring and control of the location, granularity, and persistence of agent state, for effective management of agent lifecycles, and for supporting very long-lived agent-based processes.
– To provide tool support for animation of agent logic and for unit testing.
– To develop well-founded service-oriented agent interaction protocols.
– To achieve an effective integration of agents with transaction processing.
– To provide better support for the entire agent application development cycle, especially for runtime analysis and debugging.

Goal Representation for BDI Agent Systems

Lars Braubach[1], Alexander Pokahr[1], Daniel Moldt[2], and Winfried Lamersdorf[1]

[1] Distributed Systems and Information Systems
Computer Science Department, University of Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
{braubach, pokahr, lamersd}@informatik.uni-hamburg.de
[2] Theoretical Foundations of Computer Science
Computer Science Department, University of Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
moldt@informatik.uni-hamburg.de

Abstract. Agent-oriented system development aims to simplify the construction of complex systems by introducing a natural abstraction layer on top of the object-oriented paradigm composed of autonomous interacting actors. One main advantage of the agent metaphor is that an agent can be described similar to the characteristics of the human mind consisting of several interrelated concepts which constitute the internal agent structure. General consensus exists that the Belief-Desire-Intention (BDI) model is well suited for describing an agent's mental state. The desires (goals) of an agent represent its motivational stance and are the main source for the agent's actions. Therefore, the representation and handling of goals play a central role in goal-oriented requirements analysis and modelling techniques. Nevertheless, currently available BDI agent platforms mostly abstract from goals and do not represent them explicitly. This leads to a gap between design and implementation with respect to the available concepts. In this paper a generic representation of goal types, properties, and lifecycles is developed in consideration of existing goal-oriented requirements engineering and modelling techniques. The objective of this proposal is to bridge the gap between agent specification and implementation of goals and is backed by experiences gained from developing a generic agent framework.

1 Introduction

When designing and building agent applications the developer is confronted with several intricate issues, ranging from general aspects such as development processes and tools to concrete design decisions like how agents should act and interact to implement a certain application functionality. These issues are addressed in the Jadex research project,[1] which aims to provide technical and conceptual support for the development of open multi-agent systems composed of rational

[1] `http://vsis-www.informatik.uni-hamburg.de/projects/jadex`

and social agents. One main topic of the project is reviewing and extending concepts and software frameworks for developing goal-directed agents following the BDI model. With respect to goals in agent systems the topic poses several interesting questions, which can be categorised into representational, processing, and deliberation related issues.[2]

Representation:
1. *Which generic goal types and properties do exist?*
2. *Which goal states do exist during a goal's lifetime?*
3. Which structures can be used to represent goal relationships?

Processing:
4. *How does an agent create new goals and when does it drop existing ones?*
5. How does an agent reason and act to achieve its goals?
6. Which mechanisms do exist to delegate goals to other agents?

Deliberation:
7. *What are the possible agent's attitudes towards its goals?*
8. How can an agent deliberate on its (possibly conflicting) goals to decide which ones shall be pursued?

In the following the meaning of these questions will be shortly sketched. Regarding the representational aspect it is of interest which classifications of goals exist and which generic types of goals can be deduced from the literature and from implemented systems. Additionally, it is relevant which properties are exhibited by goals in general and specific goal types in particular. The second question refers to the goal lifecycle regarding the fact that goals can be in different states from the agent's point of view. On the one hand goals may differ in the agent's attitude towards them (see also question seven). This means that an agent e.g. sees some of its goals merely as possible options, which are currently not pursued in favour of other goals, and sees others as active goals, which it currently tries to achieve. On the other hand the goals may expose different processing states with respect to their type and achievement state. The third point focuses on the relationships between goals themselves, and between goals and other concepts. Relationships between goals are used for goal refinement purposes and for deliberation issues by making explicit how one goal (positively or negatively) contributes to another goal. The relationships to other concepts mainly influence creation and processing of goals, as discussed by the next two questions.

The aspect of goal processing comprises all mechanisms for goal handling during execution time. The initial question is how an agent comes to its goals and in what situations it may drop existing goals [11, 19, 23]. Intimately connected with this issue are deliberation aspects like the goal and intention commitment strategies, which define the degree of reconsideration an agent exposes. Extensive considerations about different intention commitment strategies can be found in descriptions of the IRMA agent architecture [6, 27]. Secondly, it is of importance which mechanisms an agent can use to try to achieve its goals. The process of

[2] This paper focuses on the *emphasised* questions.

plan selection and execution is a key element of BDI architectures and requires addressing further questions: How can the applicable plans be determined? Shall applicable plans be executed in parallel or one at a time? What mechanisms shall be used for the meta-level reasoning to select a plan for execution from the set of applicable plans? Partly, these questions are answered by proposed BDI architectures [6, 29] and by implemented systems [15, 16]. A complete discussion about the problems of this topic can be found in [8]. An important point of plan execution is that the agent should be able to recover from plan failures and have the possibility to try other means to achieve the goal it has itself committed to. Hence, a declarative goal representation would help to decouple plan from goal success resp. failure [37]. Another interesting point concerns goals in multi-agent systems (MAS) e.g. how an agent can delegate tasks to other agents. Goal delegation is one possibility of how this can be achieved. The topic has to address, besides the semantic meaning of goal delegation, issues of commitment, trust, and organisational structures [2, 20, 31].

Goal deliberation is part of the whole deliberation process, which comprises all meta-operations on the agent's attitudes such as belief revision and intention reconsideration. It is concerned with the manipulation of the goal structure of an agent, i.e. goal deliberation has the task to decide which goals an agent actively pursues, which ones it delays, and which ones it abandons. Necessary requirement for a goal deliberation mechanism to work is that the agent's attitudes towards its goals are clearly defined. Currently no general consensus exists how goal deliberation can be carried out. Instead, several approaches exist that address the topic with different strategies. The agent language 3APL introduces meta-rules for all of the agent's attitudes, which are executed during the interpreter cycle [10]. In contrast to this rule-based approach KAOS and Tropos allow the direct specification of contribution relationships between goals which form a basis for the decision process [9, 14]. In [33, 34] a mechanism based on pre- and post conditions for plans and goals is proposed and evaluated.

Considering these questions it is rather astonishing that available BDI multi-agent platforms such as JACK [15], JAM [16], or Jason [4] do not use explicit goal representations and therefore cannot address most of the aforementioned topics. One reason for this shortcoming is that most actual systems are natural successors of the first generation BDI systems (PRS [17, 13] derivates), which had to concentrate on performance issues and do without computationally expensive deliberation processes due to scarce computational resources. Additionally, the actual systems are mostly based on formal agent languages like AgentSpeak(L) [28] which focus on the procedural aspects of goals and treat them in an event-based fashion.

Nevertheless, the need for explicit goal representation is expressed in several recent publications [36, 37] and is additionally supported by the classic BDI theory, which treats desires (possibly conflicting goals) as one core concept [5]. The importance of explicit and declarative goal representation in the modelling area is underlined by BDI agent methodologies like Prometheus [24], Tropos [14] and requirements engineering techniques like KAOS [9, 18]. Additionally, Winikoff et al. state in [37] "[...] by omitting the declarative aspect of goals the ability to reason about goals is lost", what means that the representation of goals is a necessary precondition when one wants reasoning about goals to become possible. Therefore, we claim that the usage of explicit goals should be extended from analysis- and design- to the implementation-level. Additionally, we think that this representation issue can be generalised and that one main objective of agent-oriented software development should be to support the continuity of concepts during the requirements, analysis, design, and implementation phase. This allows preserving the original abstraction level as far as possible throughout the development phases [21].

In this paper mainly generic goal representation issues for agent-oriented programming will be discussed with respect to the existing approaches coming from the requirements engineering and modelling area and from implemented systems. In the next section an example scenario is presented. Thereafter a generic model and lifecycle for goals is proposed and validated with respect to the given scenario in section 3. The model is elaborated further on to derive more specific goal types and representations. In section 4 the implementation of the proposed goal concepts for the Jadex agent system is sketched and finally, it is shown in section 5 that the concepts are well suited to be used in practical implementations by demonstrating how the example scenario can be realised. A summary and an outlook conclude the paper.

2 Example Scenario

In this section, a derivation of the so-called "cleaner world" scenario is described. It is based on the idea that an autonomous cleaning robot has the task to clean up dirt in some environment. This basic idea can be refined with respect to various aspects and already forms the foundation for several discussions about different agent and artificial intelligence topics (e.g. in [3, 12, 28, 30]).

In our scenario of the cleaner world the main system objectives are to keep clean a building at day, e.g. a museum, and to guard the building at night. To be more concise we think of a group of cleaning robots that are located in the building and try to accomplish the overall system goals by pursuing their own goals in coordination with other individuals. Therefore, four key goals for an individual cleaning robot were identified. First, it should clean its environment at day by removing dirt whenever possible. The cleaning robot therefore has to pick-up any garbage and carry it to a near waste bin. Secondly, it has to guard the building at night by performing patrols that should be based on varying routes. Any suspicious occurrences that it recognises during its patrols should be reported to some superordinated authority. Thirdly, it should keep operational by monitoring its internal states such as the charge state of its battery or recognised malfunctions. Whenever its battery state is low it has to move to the charging station. Fourthly, the robot should always be nice to other people that are close-by. This means that it should not collide with others and greet when this is appropriate.

These top-level goals of a cleaner agent can be further decomposed to more concrete subgoals. For example to clean up a piece of waste the robot first has to move to the waste and pick it up. Then it has to find a waste bin, move to the waste bin's location, and drop the waste into it. Similar refinements also apply to the other top-level goals.

3 Modelling Goals

The importance of goal representation is reflected through a variety of proposals for goal descriptions during the requirements acquisition, analysis, and design phases. In [35] three different kinds of goal criteria are stated that correspond to the distinctive features one would naturally deduce when considering a goal as a first class object; namely the object's type, the object's attributes, and the object's relations to other objects.

First characteristic is the goal type for which different taxonomies exist, which emphasise miscellaneous aspects. *System goals* represent high-level goals the software system needs to achieve to fulfil the system requirements and can be opposed to *individual goals* of single actors in the setting [9]. Another goal type distinction is made between so-called *hard* and *soft goals* [35]. Hard goals describe services the system is expected to deliver whereas soft goals refer to non-functional properties such as the expected system qualities like performance or excellence issues.

A very important classification relates to the *temporal behaviour of a goal* and additionally fits to the way in which humans tend to think and talk about goals. This classification is especially important for the design and implementation of agent based software, as it provides an abstraction for certain generic application behaviour. For example, a so-called *achievement goal* represents the common natural understanding of the word 'goal' as something to be achieved [9, 37]. In contrast, a *maintenance goal* is introduced to observe and maintain some world state as long as the goal exists [9].

Second characteristic of goals are their attributes that consist of properties relevant for all types of goals like name, description, priority, and other attributes that are type specific such as the target state specification for an achievement goal. Furthermore, goals can exhibit an arbitrary number of application specific attributes that are directly related to the problem domain like the desired location as part of a movement goal. Additionally, for implementing the Jadex BDI system several general goal properties were identified that are important for the interpretation of goals in the running system. Contrary to goals in natural language, which bear on a huge amount of implicit context and background knowledge, the semantics of executable goals, like the exclusion or retry mode for plan selection, has to be defined exactly [25].

Third characteristic of goals are their relationships to other objects, in first consequence to other goals. Such relationships between goals are typically hierarchical goal structures, which highlight refinement relationships with respect to the used refinement strategy. A common strategy used in several modelling approaches are the AND/OR graphs [22]. An AND-refined goal demands that all its subgoals become satisfied while an OR-refined goal is fulfilled when at least one of the alternative subgoals is reached. An extensive discussion about goal relationships can be found in [35].

When talking about goals as objects it becomes apparent that they do not only exhibit these different characteristics, but additionally they need to be created in a suitable moment in the context of some actor to whom they belong. Only when new goal instances are generated during an agent's lifetime the agent will show rational behaviour in the sense that it proactively pursues its ideas [11]. And only when it exactly knows which goals actually exist and how the goals are interrelated, some deliberation mechanism can guide the agent to decide which goals should be pursued. We will now go on to discuss these issues with respect to the example scenario, thereby developing an explicit goal model and lifecycle.

3.1 Lifecycle

In the cleaner world scenario different goals can be identified for a cleaning agent. We will start our discussion with the cleanup-dirt goal, as it most closely matches the goal concepts commonly found in the literature. The desired behaviour of the agent is to pick up dirt whenever it sees it. This includes the statement of what to do (pick up dirt) and when to do it (sees dirt). Once the agent has achieved the goal, it can drop its intention towards it. To represent this goal in an agent application the developer should be able to specify in addition to the state to achieve, the condition (called production rule in [11]) when this goal should be created, therefore giving an answer to the question how an agent derives new goals.

When it notices some dirt in the environment and cannot clean-up the waste at the moment, e.g. because it already carries waste to the waste bin, it should be capable of memorising the new dirt positions to come back later and remove the litter. Hence, it should be able to form new still inactive clean-up goals (options) that should become active as soon as it is appropriate. Assuming that the environment changes during a time the agent cannot observe this area the agent might pursue a goal that is not appropriate any longer, e.g. some rubbish is blown away by the wind and the agent heads towards the memorised but outdated waste position. As soon as it can see the target position, it will notice that the waste has vanished and should drop the clean-up goal. Therefore, in addition to the conditions for goal creation, the representation of goals should allow the specification of the conditions under which a goal should be dropped.

In contrast to the cleanup-dirt goal, which is created and later dropped for each piece of waste, other goals (e.g. look for dirt, patrol) would be directly given to the agent when it is born and should persist during the lifetime of the agent. It can be noted that, although it is natural to say that the agent has both of these goals, only one of these goals is actively pursued depending on the daytime. Therefore, when representing such goals, the agent developer has to specify the context in which the goal should be pursued (e.g. day or night). Another thing that has to be captured by the goal representation is the fact that when the agent sees some dirt it will form a new cleanup goal, which should be prioritised over the look-for-dirt goal. The agent should stop wandering around searching for dirt and cleanup the dirt it has found immediately. Therefore, the agent should be able to deliberate about its current goals to decide which one should be actively pursued and which ones should be dropped or inactivated (made to an option).

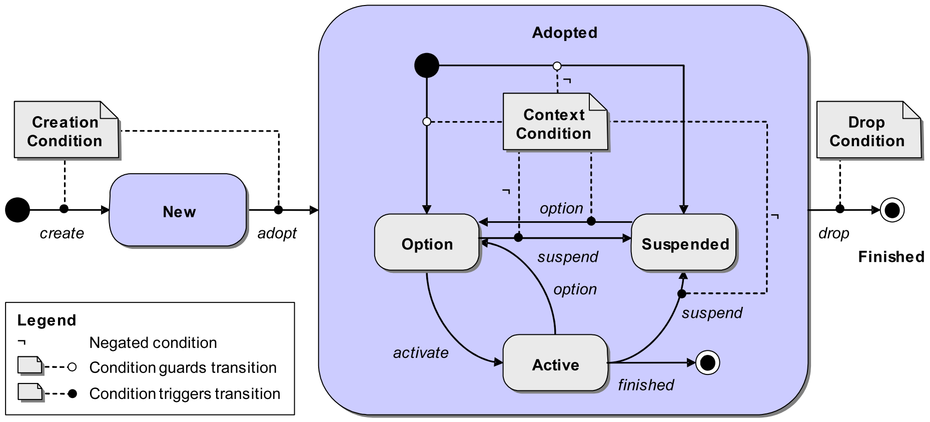


Fig. 1. Goal lifecycle

In Fig. 1 a proposal for a generic goal lifecycle that meets the requirements mentioned above is depicted in a UML statechart like fashion. It is shown that a goal can be in the states *New*, *Adopted* or *Finished*. The initial state of a newly created goal is *New*, what means that the goal exists as an idea but is not yet considered by the agent's deliberation mechanism. Therefore, the agent has to adopt the goal to pursue its new objective. By any means, the agent can always decide not to pursue the goal any more and drop it. The transitions between the different states can be either forced (not part of goal specification), e.g. a plan could create a new goal or drop a subgoal, or can be monitored by so-called conditions (specified as part of a goal). Conditions are annotated to several state transitions in two different ways to express either that the condition is used as a guard for the corresponding transition or that it represents the transition's trigger (see legend of Fig. 1). This means that a goal instance is created and adopted every time when the creation condition of this goal fires. Accordingly, it is dropped when its drop condition triggers.

Most interesting is the complex *Adopted* state which consists of the substates *Option*, *Active*, and *Suspended*. Adopting a goal makes this goal desirable to achieve for the agent and adds it to the agent's desire structure. The goal can be seen as an option that could possibly be pursued when the actual circumstances allow this. To be actively pursued the agent's deliberation mechanism has to activate the goal and so initiate the goal processing. The deliberation mechanism can also deactivate the goal at any time by moving the goal to the option state again. Whenever the goal is an option or is active it can be suspended when the goal's context becomes invalid which is indicated by the goal's context condition. Here, a negation sign at the connection between condition and state transition indicates that the inverse of the condition is used as trigger for the transition. The suspension holds as long as the context stays invalid. A suspended goal is not actively pursued similar to an option, but in contrast to an option it cannot be activated by the deliberation mechanism due to its invalid context. When the context becomes valid again the goal is made an option to allow the deliberation component to reactivate the goal whenever appropriate.[3]

3.2 Types of Goals

As already mentioned, an important classification can be made with respect to the temporal behaviour of a goal. Unfortunately, there is no single exact set of suitable types of goals that can be used. Rather a multitude of different specifications and notions emerged from different sources such as methodologies or implemented systems (see Table 1).

Table 1. Several Different Goal Types

	KAOS	Gaia	JACK	PRS	JAM	Jadex
achieve	x	x	x	x	x	x
maintain	x	x		x	x	x
cease	x					
avoid	x					
optimise	x					
test			x	x		
query					x	x
perform					x	x
preserve			x	x		

The KAOS goal-oriented requirements engineering framework [9, 18] includes the already mentioned *achieve* and *maintain* goals. Additionally, KAOS introduces the negation of the aforementioned types called *cease* (as opposed to achieve) and *avoid* (as opposed to maintain). These two types of goals ease the description of goals in a natural way and semantically they can be traced back to the original types [35]. Furthermore, *optimise* goals for maximising or

[3] An interesting analogy to the goal lifecycle can be found in the operating systems area. The substates (*option*, *active*, *suspended*) of the adopted state resemble the states *ready*, *running*, *blocked* known from operating system processes [32]. Just like a blocked process, a suspended goal cannot be directly reactivated. In both cases a higher-level authority (the OS scheduler resp. the agent's deliberation mechanism) is responsible for selecting among the available options.

minimising some target value are proposed. The well-known Gaia methodology [38] does not introduce any goals at all, but uses liveness and safety properties for roles. Liveness properties describe states the agent has to bring about, whereas safety properties specify system invariants. In this way they are comparable with the achieve and maintain goal semantics.

The JACK agent system [15] offers in addition to *achieve* goal semantics the *test* and *preserve*[4] goal types. A test goal can be used to find out if a condition holds and a preserve goal is the passive version of a maintain goal in the sense that the goal controls a state and vanishes when this state is violated. In contrast to JACK, the C-PRS system [17] supports maintain goals at the implementation level. Besides *achieve* and *maintain* goals, the JAM interpreter [16] and the Jadex system [26] support *query* goals, which are similar to achieve goals. Query goals allow for an easy information retrieval from the beliefbase and when the result is not available the BDI mechanism will invoke plans for retrieving the needed information. The fourth type of goal that JAM and Jadex support is the *perform goal*, which is not related to some desired world state but to an activity. It ensures that an activity will be done in some future state [16].

In the rest of this paper we will concentrate on the *perform, achieve*, *query,* and *maintain* goal types. From Table1 one can see that the *achieve* and the *maintain* goal types are especially important, because they are in widespread use. *Cease* and *avoid*, on the other hand, exhibit the same execution semantics as *achieve* resp. *maintain.* The *optimise* goal belongs to the class of soft goals, which is outside the scope of this paper [35]. The *perform* goal is interesting, because it does not refer to a world state being achieved or maintained but to activities that should be performed. *Test* and *query* goals serve the same purpose to describe information acquisition, therefore only one of them is considered. Finally, The *preserve* construct is merely called a goal. In fact, it represents just a guarded action [17].

The following sections take a closer look at those interesting types of goals and their corresponding properties. Unlike the pure modelling approaches (KAOS, Gaia) it will be made explicit how goals following these models are processed at runtime using a BDI interpreter. One important aspect is therefore how their execution semantics relates to the generic goal lifecycle presented above. This is handled in a general way by the refinement of the active state, which reveals special information about the type of goal for goal processing. The example scenario is used as an evidence for the presented properties and behaviour, where appropriate.

Perform Goal. A *perform goal* specifies some activities to be done, therefore the outcome of the goal depends only on the fact if activities were performed [16]. Naturally, when no activities could be performed, e.g. because no plan was applicable in the actual context, the goal has *Failed.* Otherwise, when one or more plans have been executed the goal can enter the *Succeeded* state.

[4] It adds to the confusion about goal types that in JACK the *preserve* behaviour is obtained using the `@maintain` keyword.

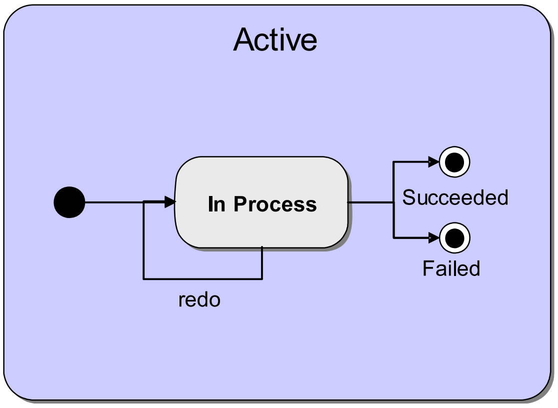


Fig. 2. Perform goal states

The refined active state of a perform goal is shown in Fig. 2. After being activated, the *In Process* state is entered, which triggers the internal plan selection and execution mechanism of the agent [26]. While plans are executing the goal stays in the *In Process* state. When the plan execution is done, i.e. no more plans are running or waiting for events, the *In Process* state is exited.

In the cleaner world example the two goals *patrol* and *do-greeting* should be modelled as perform goals, as they do not directly refer to a desired target state. While the *do-greeting* goal is finished once the greeting is performed, the *patrol* goal should not end when a patrol round is finished. Instead, the agent should continuously start new patrol rounds while the *patrol* goal is active. The *redo* property is an extension of the original JAM perform goal [16] and allows specifying that the activities of the goal should be performed iteratively. Therefore, when leaving the *In Process* state two state transitions may occur depending on the *redo* property. When *redo* is specified the goal re-enters the *In Process* state to re-start plan execution. When *redo* is not specified the goal enters one of the end states (*Failed*, *Succeeded*) causing the *Active* state to end. Looking back at the generic goal lifecycle (Fig. 1) one can see that exiting the *Active* state also causes the *Adopted* state to end (*finished* transition). Therefore, once the processing of the perform goal has stopped the goal is no longer adopted by the agent, because it is already reached or failed.

Achieve Goal. An *achieve goal* represents a goal in the classical sense by specifying what kind of world state an agent wants to bring about in the future. This target state is represented by a *target condition*. When an agent obtains a new achieve goal that shall be pursued (e.g. a cleanup goal) the agent starts activities for achieving the target state (e.g. no waste at given location). When the target state is already reached before anything has been done the goal can be considered as succeeded. Otherwise, for a yet unachieved goal the BDI mechanism is started and plans are selected for execution. Whenever during the plan execution phase the target condition switches to success all running plans of that goal can be aborted and the goal is reached. In [37] the description of an achieve goal is enriched with an additional failure condition, which helps to terminate the goal when it is absolutely not achievable any more. The difference to the drop condition introduced in the generic goal lifecycle is that the drop condition does not determine the final state of the goal. In contrast, the failure condition indicates that the agent is unable to achieve the goal and therefore the goal has failed.

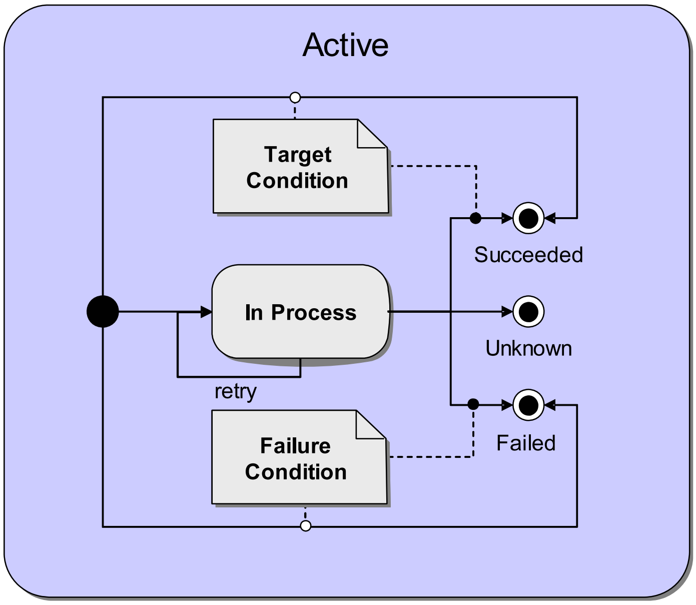


Fig. 3. Achieve / Query goal states

Fig. 3 shows the specific behaviour of an achieve goal. The main difference to the perform goal type is the target condition that specifies the desired world state to be achieved. An activated achieve goal will first check its target condition for fulfilment and enter the succeeded state directly when nothing needs to be done. Additionally, the failure condition will be checked to abort the goal when the condition is true. When none of them has fired the goal will enter the *In Process* state to start the execution of applicable plans. In contrast to the perform goal, plan execution may be terminated at any time when the target or failure condition become satisfied. In this case the goal is finished and moves to the *Succeeded* resp. *Failed* state.

When there are no more plans to execute and none of the executed plans could be completed successfully the goal moves to the *Failed* state. Another final state *Unknown* is entered when the execution is finished, some plans have been executed properly, but the agent cannot determine the truth-value of the target condition (e.g. due to insufficient knowledge). Any of the three final states will cause the *finished* transition of the generic goal lifecycle (Fig. 1) to trigger. For example, when the given location is clean a cleanup goal is succeeded and can therefore be removed from agent's goal structure.

Query Goal. A *query goal* is used to enquire information about a specified issue. Therefore, the goal is used to retrieve a result for a query and does not necessarily cause the agent to engage in actions. When the agent has sufficient knowledge to answer the query the result is obtained instantly and the goal succeeds (e.g. an agent wants to find a waste bin and already knows the location). Otherwise, applicable plans will be tried to gather the needed information (e.g. searching for a waste bin).

The underlying model of the query goal resembles to a high degree the achieve goal [16]. The states of both goals are equal and are depicted in Fig. 3. Main difference between both goal types is that the query goal requires an informational result, which is captured by an implicit target condition testing if a result is available.

Maintain Goal. A *maintain goal* has the purpose to observe some desired world state and the agent actively tries to re-establish this state when it is violated. The perform, achieve, and query goal types represent goals that continuously cause the execution of plans while they are active. In contrast, an activated maintain goal may not instantly cause any plan to be executed. Fig. 4 shows that the maintain goal stays in the *Idle* state until the maintain condition is violated. Another difference is that there is no final state. Even when the *maintain condition* is currently satisfied the agent always has to monitor the environment for changes that may violate the condition. The maintain goal therefore always moves back to the *Idle* state when processing has been finished successfully.

In case the processing fails but the agent has no more applicable plan to execute, the *Unmaintainable* state is entered, which means that the agent knows that the condition is violated, but there is nothing it can do about it. Similar to the achieve goal, a maintain goal may be in the *Unknown* state when the agent cannot determine if the plan execution leads to the desired results. From the *Unknown* state a transition back to the *In Process* or *Idle* state may be done when the agent can determine the state of the maintain condition. From both the *Unknown* and the *Unmaintainable* state, the goal may periodically reenter the *In Process* state to try out if the goal can be maintained now. This behaviour is obtained by specifying the *recur* flag. In contrast to the *retry* flag, which manages the sequential execution of applicable plans, the *recur* flag leads to a complete restart of goal processing, thereby again considering previously excluded plans.

Using the maintain condition alone may sometimes lead to undesirable behaviour, because of the event driven nature of goal processing in BDI agents. Consider the *maintain-battery-loaded* goal of the cleaner agent: When the condition to be maintained is specified as `'chargestate<20%'` the agent will move to

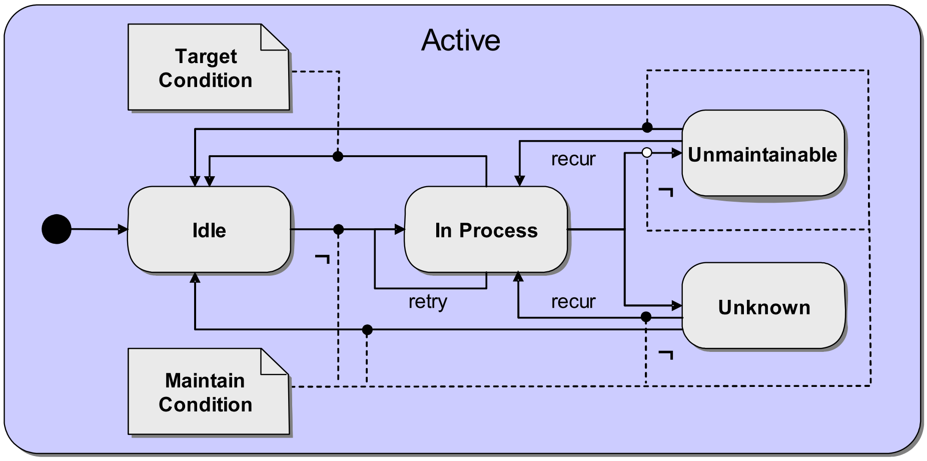


Fig. 4. Maintain goal states

the charge-station whenever the energy level drops below 20%. However, as soon as the level is back at 20% the agent will stop loading its battery, because the condition is satisfied again. Therefore, it is sometimes necessary to concretise the condition to be established whenever the maintain condition is triggered. In our model this can be specified by an optional *target condition*, which specifies when the transition to the idle state is allowed. The semantics of this extended type of maintain goal is therefore: Whenever the maintain condition is violated select and execute plans in order to establish the (more specific) target condition. In the example the maintain condition 'chargestate > 20%' can be refined to the target condition 'chargestate=100%' to make sure that the cleaner agent will always do a full recharge.

All of the specific types of goals (perform, achieve, query, maintain) inherit the same generic lifecycle presented in section 3.1. Therefore, in addition to the properties specific to a goal type (such as failure condition for achieve goals) the specification of any goal can be enriched by the generic goal properties such as creation, context, and drop condition. This makes it possible e.g. to easily specify a maintain goal that should only be pursued in a given context.

4 Goal Realisation in Jadex

The last section presented a generic model for goals in BDI agents and identified four goal types with distinct execution behaviour. In the following we will shortly sketch how this execution behaviour is realised in the generic agent framework Jadex. The next section will then show how applications like the cleaner example scenario can be easily implemented when such an abstract goal representation is available at the implementation level.

The Jadex agent framework [26, 7] is built on top of the JADE plattform [1] and provides an execution environment and an API to develop agents using beliefs, goals, and plans as first class objects. Jadex adopts well established application development technologies such as XML, Java, and OQL to facilitate an easy transition from conventional object-oriented programming to BDI agent programming.

To implement an agent the developer has to create two types of files: One XML file is used to define the agent by declaratively specifying among other things the beliefs, goals, and available plans. In addition to this agent definition file (ADF), for each plan used by the agent the plan body has to be implemented in a separate Java class. Plan implementations may use the Jadex API e.g. to send messages, manipulate beliefs, or create subgoals (for details see [25]). An expression language is used throughout the ADF to establish the connection between the declarative elements in the ADF and the object-oriented plan implementations. The language follows a Java syntax, but is extended to support OQL constructs for querying the belief base.

The goal tags in the XML file are read by the interpreter to create instances of the goals, which implement the state machines presented in section 3. The instatiated goal objects themselves take care of their lifecycle by throwing so called goal-events (leading to the execution of plans) whenever they enter the *In Process* state and by automatically performing the corresponding state transitions when goal conditions are triggered or the execution of a plan has finished. The goal conditions and parameters, which are evaluated at runtime, are specified using the Java/OQL like expression language.

At runtime the system keeps track of the instantiated goals, which may be created either as independent top-level goals or dispatched as subgoals inside of a plan. Goal processing is initiated whenever the active state of a goal is entered. Before a goal is reached, several plans may try to process the goal, even at once, when specified so. Thereby, plans only have access to a copy of the original goal object called *process* goal, to ensure a level of isolation between running plans and their associated subgoal-hierarchies. When the active state of a goal is exited (e.g. because the goal is suspended), all associated process goals are dropped leading to a termination of the corresponding plans and subgoals created by those plans. For each goal, a history of process goals is kept to remember the executed plans together with the outcome. This information is used to determine plans which should be excluded from the applicable plan list, when the goal needs to be processed again.

5 Example Implementation

The cleaner world scenario is realised as a simulation setting using two different kinds of agents. Besides the cleaner agents an environment agent acts as substitute for the real surrounding. Using an agent as environmental representation has the advantage that the setting can be easily distributed over a network of computers having cleaner agents working in the same environment located on different platforms.

The cleaner agents use vision and movement plans that interact with the environment agent following a domain dependent ontolology in which the relevant concepts and actions like waste, waste bin and pick-up resp. drop waste are defined. They update their internal beliefs with respect to the sensed environmental changes and request actions in the environment that may fail under certain conditions e.g. when two cleaners try to pick up the same piece of waste simultaneously.

Top-level Goals

- performpatrol
  - PatrolPlan
    - uses goal achievemoveto
- achievecleanup
  - CleanupWastePlan
    - uses goal achievepickupwaste
    - uses goal querywastebin
    - uses goal achievedropwaste
- maintainbatteryloaded
  - LoadBatteryPlan
    - uses goal querychargingstation
    - uses goal achievemoveto
- performlookforwaste
  - ExploreMapPlan
    - uses goal achievemoveto

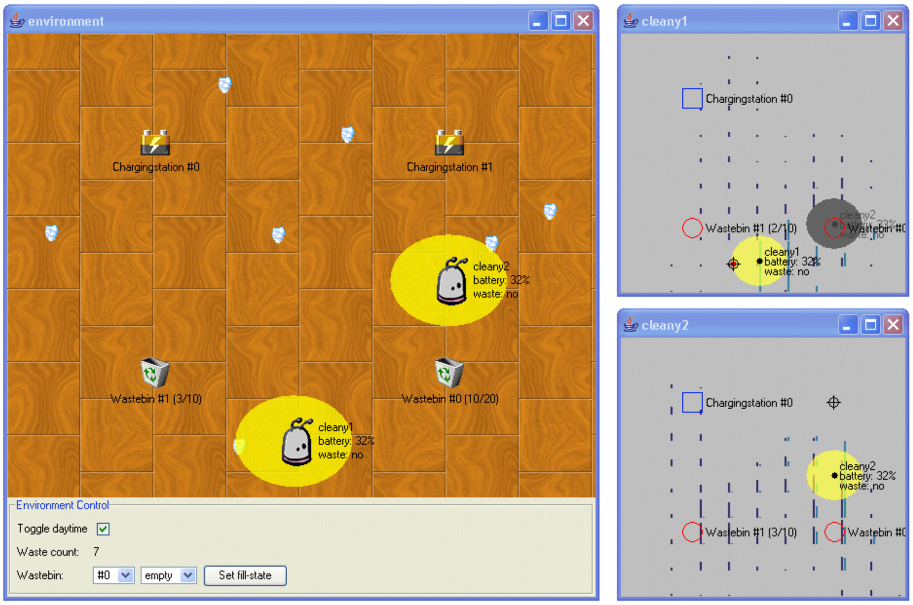
Subgoals

- achievepickupwaste
  - PickupWastePlan
    - uses goal achievemoveto
- achievedropwaste
  - DropWastePlan
    - uses goal achievemoveto
- querywastebin
  - ExploreMapPlan
    - uses goal achievemoveto
- querychargingstation
  - ExploreMapPlan
    - uses goal achievemoveto
- achievemoveto
  - MoveToLocationPlan

Fig. 5. Goal - plan overview

In Fig. 5 a brief overview of the relationships between the used goals and plans is given. On the left hand side the agents' top-level goals are shown whereas on the right hand side the subgoals that are used from within plans are depicted. For each goal at least one plan is defined that is responsible for pursuing the goal. As introduced in section 2 a cleaner agent has top-level goals for performing patrols (performpatrol), cleaning-up waste (achievecleanup) and monitoring its battery state (maintainbatteryloaded). To avoid the agent doing nothing when it currently has no duty, a goal template for searching for waste is also defined (performlookforwaste).

To handle the performpatrol goal a cleaner agent has a patrol plan that accesses a predefined route from the beliefbase and steers the agent to the actual patrol points by using the achievmoveto subgoal. Somewhat more complex is the CleanupWastePlan that is used in response to an active achievecleanup goal. It employs three different subgoals for decomposing the goal into the separate tasks of picking up a piece of waste (achievepickup), searching for a non-full waste bin (querywastebin) and finally dropping the waste into the wastebin (achievedropwaste). To be able to resume a suspended cleanup goal the plan also tests if the agent is already carrying a piece of waste. In the case that the agent already possesses the waste the pickup procedure can be omitted. For re-establishing a violated maintainbatteryloaded goal the LoadBatteryPlan tries to find a charging station, heads towards it and consumes as many energy as needed. To find a suitable station a query subgoal (querychargingstation) is used that immediately returns a result when the agent already knows a station. When this is not the case, the ExploreMapPlan is used to systematically search for a yet unknown charging station. This plan is also used in the context of the performlookforwaste goal to discover new waste in the environment.



Environment Control

Toggle daytime ✓

Waste count: 7

Wastebin: #0 empty Set fill-state

Fig. 6. Cleaner World Example Snapshot

A cleaner agent has three initial goal instances that drive its actions from birth. An instance of the performlookforwaste resp. the performpatrol goal lets the agent move around to search for waste or to observe the environment, depending on the daytime. These two goals are only active, when the agent has no other important things to do. An instance of the maintainbatteryloaded has highest priority and monitors the agent's battery state during its lifetime. In addition, several goal types are declared for goals that get instantiated and adopted under certain conditions. In the following sections some example goal declarations are explained. More implementation details can be found in the freely downloadable Jadex package, which includes a runnable implementation of the cleaner world example.[5] In Fig. 6 a snapshot of the running application is presented, which shows the global environmental view as well as the local views of two cleaner agents.

[5] available for download at `http://vsis-www.informatik.uni-hamburg.de/projects/jadex`

5.1 The Perform-Patrol Goal

Fig. 7 shows the perform-patrol goal as it is specified in the XML agent descriptor of a cleaner agent. The goal is of type *performgoal* and is given the name perform-patrol. The attribute *redo* was already introduced in the refined perform goal state chart (see Fig. 2) and causes the goal to be continuously executed as long as applicable plans are available. The *exclude* attribute is a special flag that in this case tells the BDI plan selection mechanism that plans should not be excluded from the applicable plans list once they have been executed. Therefore, the agent will continue to patrol while the goal is active using any patrol plans it has.

```
<performgoal name="performpatrol" redo="true" exclude="never">
   <contextcondition>
      !$beliefbase.is_loading && !$beliefbase.daytime
   </contextcondition>
</performgoal>
```

Fig. 7. Peform-patrol goal

The example scenario demands that the agent should only be on patrol at night. Our system does not yet capture the (positive or negative) contribution between goals, but the agent has to be prevented somehow from continuing to patrol while it tries to reload its battery. It is assumed that the agent knows if it is day or night and if its battery state is low and has to be reloaded. Using these two boolean beliefs (daytime, is_loading) the developer can specify the *contextcondition* of the goal, where $beliefbase refers to the belief base of the agent. The context condition was introduced in the generic goal lifecycle (Fig. 1) and defines when the goal can or cannot be active. The perform patrol goal may therefore only be active when the agent is not loading its battery and it is not daytime. In a similar way, a perform-look-for-waste goal is defined with a context condition that is only valid at daytime.

5.2 The Achieve-Cleanup Goal

One purpose of the cleaner agent is to remove all pieces of waste it notices. The achieve-cleanup goal (Fig. 8) is an *achievegoal* that is instantiated for every single piece of waste to clean up. The goal contains a *parameter* waste specifying which piece of waste to clean up. The given default *value* of the waste parameter is specified by a select statement that always evaluates to the piece of waste that is nearest to the agent when the goal is instantiated. The known pieces of waste (belief wastes) are sorted by distance (order by clause) to the current location (belief my_location).

For the agent to keep cleaning up every piece of waste it notices, the *creation-condition* as introduced in the generic goal lifecycle (Fig. 1) is used to trigger creation of new goal instances whenever needed. A cleanup goal will be created whenever the agent knows that there is some waste (belief wastes) and that it is not currently cleaning (belief is_cleaning). The reason for the second part of the

```
<achievegoal name="achievecleanup">
  <parameter name="waste" class="Waste">
    <value>
      select any $waste from $beliefbase.wastes order by
      $waste.location.getDistance($beliefbase.my_location)
    </value>
  </parameter>
  <creationcondition>
    $beliefbase.wastes.length>0 && !$beliefbase.is_cleaning
  </creationcondition>
  <contextcondition>
    !$beliefbase.is_loading && $beliefbase.daytime
  </contextcondition>
  <dropcondition>
    !$beliefbase.carrieswaste
    && (!$beliefbase.containsFact("wastes", $goal.waste)
    || (select any $waste from $beliefbase.wastes
      order by $waste.location.getDistance(
        $beliefbase.my_location)) != $goal.waste)
  </dropcondition>
  <targetcondition>
    (select any $wastebin from $beliefbase.wastebins
    where $wastebin.contains($goal.getParameter("waste"))) !=null
  </targetcondition>
</achievegoal>
```

Fig. 8. Achieve-cleanup goal

condition is that there is currently no deliberation mechanism telling the agent which cleanup goal to achieve first when there is more than one present at the same time. Therefore, the is_cleaning belief is used to assure that only one cleanup goal at a time is created. As with the perform-patrol goal a context condition is used to constrain under which circumstances the goal may be active: The agent should pursue cleanup goals only when it is not loading its battery and only at daytime. The goal is achieved when the waste is contained in one of the known waste bins as described in the *target condition*.

In our example implementation we also added a rather complex *dropcondition* for the cleanup goal, which is not necessary for correct operation, but helps to improve the performance of the cleaner agent. To allow opportunistic cleanup of new pieces of waste and to avoid unnecessary movement of the cleaner, an existing cleanup goal is dropped when the agent comes to know that the piece of waste to be picked up is no longer there or another piece of waste is closer to the agent.

5.3 The Query-Wastebin Goal

The query-wastebin goal shows how a goal to query for information can be realised in Jadex (see Fig. 9). Assuming that the agent does not completely know its environment, the objective of the goal is to find a waste bin that is not full and near to the agent. This goal is created by a plan as a subgoal of the achieve-cleanup goal once the agent has picked up some dirt (cf. sect. 2).

```
<querygoal name="querywastebin" exclude="never">
  <parameter name="result" class="Wastebin" optional="true">
    <value evaluationmode="on_demand">
      select any $wastebin from $beliefbase.wastebins
      where !$wastebin.isFull() order by
      $waste.location.getDistance($beliefbase.my_location)
    </value>
  </parameter>
</querygoal>
```

Fig. 9. Query-wastebin goal

It is modelled as *querygoal* and has a *parameter* result. This parameter is bound to the nearest not full waste bin, if any, and is evaluated on_demand what means that the select expression is evaluated whenever the parameter value is accessed. The *targetcondition* of the query goal is not stated and therefore the default target condition for query goals is used. Hence the goal succeeds when a result is retrieved, i.e. a not full waste bin nearby was found. The implicit target condition allows for opportunistic goal achievement (see Fig. 3), that is, the goal succeeds without the execution of any plan if the agent already knows the location of a not full waste bin.

```
<maintaingoal name="maintainbatteryloaded">
  <maintaincondition>
    $beliefbase.my_chargestate > 0.2
  </maintaincondition>
  <targetcondition>
    $beliefbase.my_chargestate == 1.0
  </targetcondition>
</maintaingoal>
```

Fig. 10. Maintain-battery-loaded goal

5.4 The Maintain-Battery-Loaded Goal

The cleaning agent has to stay operational; therefore it has to monitor its internal state and will occasionally move to the charging station to reload its battery. The specification of the maintain-battery-loaded goal is given in Fig. 10. The goal is a *maintaingoal* and therefore includes a *maintaincondition* and a *targetcondition* as present in the refined maintain goal state chart (see Fig. 4). The maintain condition monitors the battery state (belief my_chargestate) and triggers plan execution whenever the charge state drops below 20%. The refined target condition causes the battery to be always reloaded to 100% before the goal moves back to the idle state.

6 Conclusions and Outlook

This paper provides two main contributions. First, the way of how an agent attains and manages its goals is analysed and a generic lifecycle is proposed that models the different states of goals in BDI agent systems. Secondly, the generic goal lifecycle is refined into different goal types which capture commonly required agent behaviour. Both of these contributions are backed by the cleaner world example at the conceptual as well as implementation level.

The example shows that the proposed goal model is well suited for a natural description of an agent-based system. The continuous usage of abstract concepts in the design and implementation phases considerably simplifies the development of software agents compared to the current practice of using object-oriented techniques. Additionally, it helps to preserve the abstraction level throughout the whole development process. The system is easier to design, as the involved goal concepts are closer to the way that humans think and act. The transition to the implemented system is largely simplified, because only minor refinements of design specifications are necessary to obtain an executable system. Moreover, the development is less error-prone, as large portions of complex agent behaviour, such as goal creation and processing, are already implemented in the underlying agent architecture. Finally, the types of goals available in the agent language have the additional effect that they may guide the agent developer in its analysis and design decisions, because they represent a natural and abstract means for describing the application domain.

This work is also the result of practical considerations when realising the proposed goal model in an efficient and easy to use software framework. The model includes those goal types and properties that frequently occured in the researched systems and methodologies and that have practical relevance for agent systems we have built so far.

The presented goal model does not cover all important aspects of goals as they are presented in the introduction. One point that was not addressed by this paper affects the relations between goals such as hierarchies for goal decomposition. In this field, especially concerning the requirements and modelling phases, a lot of research has already been done and it has to be evaluated if these concepts can be successfully transferred to the design and implementation phase of MAS. Another important aspect of goals that was covered only marginally in this paper is goal deliberation. With the help of deliberation mechanisms, the agent is able to select between different goals, detect goal conflicts and handle them appropriately. The precondition for goal deliberation is the explicit and declarative representation of goals, which is not reflected in actual agent systems and agent languages. Therefore, the conceptualization of the introduced goal model is the foundation for further explorations of different deliberation mechanisms.

References

1. F. Bellifemine, G. Rimassa, and A. Poggi. JADE – A FIPA-compliant agent framework. In *4th Int. Conf. Practical Applications of Agents and Multi-Agent Systems (PAAM-99)*, pages 97–108, London, UK, December 1999.
2. F. Bergenti, L. Botelho, G. Rimassa, and M. Somacher. A FIPA compliant Goal Delegation Protocol. In *Workshop on Agent Communication Languages (AAMAS 2002)*, 2002.
3. R. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifiable multi-agent programs. In *Proceedings of the First International Workshop ProMAS*, pages 43–49, Australia, 2003.
4. R. H. Bordini and J. F. Hübner. *Jason User Guide*, 2004. `http://jason.sourceforge.net/`.
5. M. Bratman. *Intention, Plans, and Practical Reason.* Harvard University Press, 1987.
6. M. Bratman, D. Israel, and M. Pollack. Plans and Resource-Bounded Practical Reasoning. In *Philosophy and AI: Essays at the Interface*, pages 1–22. The MIT Press, 1991.
7. L. Braubach, A. Pokahr, and W. Lamersdorf. Jadex: A Short Overview. In *Net.ObjectDays 2004: AgentExpo*, 2004. (to be published). `http://vsis-www.informatik.uni-hamburg.de/papers/jadex_node.pdf`.
8. P. Busetta, N. Howden, R. Rönnquist, and A. Hodgson. Structuring BDI Agents in Functional Clusters. In *Intelligent Agents VI, ATAL '99*. Springer, 2000.
9. A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1–2):3–50, April 1993.
10. M. Dastani, F. de Boer, F. Dignum, and J.J. Meyer. Programming Agent Deliberation: An Approach Illustrated Using the 3APL Language. In *Proceedings of AAMAS'03*, 2003.
11. Frank Dignum and Rosaria Conte. Intentional Agents and Goal Formation. In *Agent Theories, Architectures, and Languages*, pages 231–243, 1997.
12. J. Firby. An Architecture for A Synthetic Vacuum Cleaner. In *Proc. of the AAAI Fall Symp. Series Workshop on Instantiating Real-World Agents*, Raleigh, NC, October 1993.
13. M. Georgeff and A. Lansky. Reactive Reasoning and Planning: An Experiment With a Mobile Robot. In *Proceedings of the 1987 National Conference on Artificial Intelligence (AAAI 87)*, pages 677–682, Seattle, Washington, July 1987.
14. F. Giunchiglia, J. Mylopoulos, and A. Perini. The Tropos Software Development Methodology: Processes, Models and Diagrams. In *Proc. of AAMAS02*. ACM Press, 2002.
15. N. Howden, R. Ronnquist, A. Hodgson, and A. Lucas. JACK Intelligent Agents - Summary of an Agent Infrastructure. In *Proc. 5th ACM Int. Conf. on Autonomous Agents*, 2001.
16. M. Huber. JAM: A BDI-Theoretic Mobile Agent Architecture. In *3rd Annual Conf. on Autonomous Agents (AGENTS-99)*, pages 236–243, New York, May 1–5 1999. ACM Press.
17. F. Ingrand, R. Chatila, R. Alami, and F. Robert. PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots. In *Proc. of the IEEE Int. Conf. on Robotics and Automation*, pages 43–49, Minneapolis, April 1996.
18. E. Letier and A. van Lamsweerde. Deriving operational software specifications from system goals. In *Proc.of the 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 119–128. ACM Press, 2002.

19. M. Luck and M. d'Inverno. Motivated Behaviour for Goal Adoption. In *Multi-Agent Systems: Theories, Languages and Applications - 4th Australian Workshop on Distributed Artificial Intelligence*, pages 58–73. Springer-Verlag, 1998.
20. Á. Moreira, R. Vieira, and R. Bordini. Extending the operational semantics of a BDI agent-oriented programming language for introducing speech-act based communication. In *Proc. Declarative Agent Languages and Technologies (DALT-03), held with AAMAS-03*, 2003.
21. J. Mylopoulos. Requirements-Driven Information Systems Development. Invited Talk, AOIS'99 at CAiSE'99, Heidelberg, Germany, 1999.
22. N. Nilsson. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, 1971.
23. T. J. Norman and D. Long. Goal creation in motivated agents. In *Intelligent Agents, Proc. of ATAL'95*, pages 277–290. Springer-Verlag, 1995.
24. L. Padgham and M. Winikoff. Prometheus: A methodology for developing intelligent agents. In *3rd Int. Workshop on Agent Oriented Software Engineering (AOSE02)*, July 2002.
25. A. Pokahr and L. Braubach. *Jadex User Guide*, 2004. `http://vsis-www.informatik.uni-hamburg.de/projects/jadex/download.php`.
26. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: Implementing a BDI-Infrastructure for JADE Agents. *EXP – in search of innovation*, 3(3):76–85, 2003.
27. M. Pollack. The Uses of Plans. *Artificial Intelligence*, 57(1):43–68, 1992.
28. A. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In *7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, 1996.
29. A. Rao and M. Georgeff. BDI Agents: from theory to practice. In *Proc. of the 1st Int. Conference on Multi-Agent Systems (ICMAS'95)*, pages 312–319. The MIT Press, 1995.
30. S. Russell and P. Norvig. *Artifical Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1995.
31. M. Somacher, M. Tomaiuolo, and P. Turci. Goal Delegation in Multiagent System. In *Proc. Tecniche di Intelligenza Artificiale per la ricerca di informazione sul Web, Siena*, 2002.
32. A. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, 2001.
33. J. Thangarajah, L. Padgham, and M. Winikoff. Detecting and Avoiding Interference Between Goals in Intelligent Agents. In *Proceedings of IJCAI 2003*, August 2003.
34. J. Thangarajah, L. Padgham, and M. Winikoff. Detecting and Exploiting Positive Goal Interaction in Intelligent Agents. In *Proceedings of AAMAS'03*, 2003.
35. A. van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *Proc. RE'01 - Int. Joint Conference on Requirements Engineering*, pages 249–263. IEEE, 2001.
36. M. Winikoff, J. Harland, and L. Padgham. Linking Agent Concepts and Methodology with CAN. `http://citeseer.ist.psu.edu/497423.html`.
37. M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative & Procedural Goals in Intelligent Agent Systems. In *Proc. of KR03*. Morgan Kaufmann Publishers, 2002.
38. M. Wooldridge, N. Jennings, and D. Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

AF-APL[⋆] – Bridging Principles and Practice in Agent Oriented Languages

Robert Ross[1], Rem Collier[2], and G.M.P. O'Hare[2]

[1] FB3, Mathematik-Informatik, Universität Bremen, Germany
robertr@tzi.de
[2] Department of Computer Science,
University College Dublin, Ireland
{rem.collier, gregory.ohare}@ucd.ie

Abstract. For AOP (Agent Oriented Programming) to become a mature discipline, lessons must be learned from practical language implementations. We present AF-APL (AgentFactory - Agent Programming Language) as an Agent Oriented Programming Language that has matured with continued revisions and implementations, resulting in a language - which, although based on the more theoretical aspects of AO design - has incorporated many of the practical considerations of programming real world agents. We describe AF-APL informally, focusing on its experience driven features, such as commitment reasoning, a rich plan operator set, and an inherent asynchronous design. We present the default execution cycle for the AF-APL interpreter, looking in detail at the Commitment Management model. This model provides an agent with power to reason about its own actions, while maintaining basic constraints on computational tractability. In our development of the language, we learned many lessons that are not covered in the purer AO language definitions. Before concluding, we discuss a number of these lessons.

1 Introduction

Agent-Oriented Programming (AOP) represents one approach to the fabrication of agent-oriented applications. It is based upon the premise that complex agent behaviours can be best represented in terms of a set of mental notions (e.g. belief, goal, and commitment), and the interplay between them.

Research in this area has produced a number of agent programming languages, including Agent0 [1], 3APL [2], AgentSpeak(L) [3], and AF-APL [4].

[⋆] As of the date of publication, the name of the language presented in this paper has changed from AF-APL to ALPHA (A Language for Programming Hybrid Agents). This change reflects our wish to highlight differences between this language and its ancestor, originally presented in [4]. This paper retains the name used during the submission and review process, but any subsequent publications and documentation shall use the name 'ALPHA'.

Many of these languages draw upon earlier work on logical agent theories that use Possible Worlds semantics [5] [6] [7]. However, computation tractability issues have led to a number of them being re-specified using more tractable approaches such as formal methods (e.g. VDM and Z) and operational semantics.

While the use of formal semantics has resulted in AOP languages that are conceptually well defined, many of them have not been applied to large scale projects. These projects raise a number of practical issues regarding the use of AOP languages, such as usability, scalability, and applicability.

It is these issues that have driven the work presented in this paper. Specifically, we describe an extended version of AF-APL [4], an AOP language that was originally derived from a logical model of commitment. These extensions, are rooted in practical experiences gained during the last 3 years. Specifically, AF-APL has been employed in the development of a number of large scale agent-oriented applications in the robotics [8] and mobile computing [9] domains. These experiences have driven a number of extensions to AF-APL that include: the introduction of goals, additional plan operators, a new commitment management system, support for commitment reasoning, and reactive rules.

The rest of the paper is structured as follows: Section 2 presents a review of significant work to date in AO language development. This is followed in Sections 3 and 4 with a description of AF-APL - the first of these sections details AF-APL's constructs, while Section 4 looks at the AF-APL execution model. Although this paper focuses on the AF-APL language, we have presented a basic overview of an AF-APL interpreter; That interpreter, based on the AgentFactory Framework, is presented in Section 5. Section 6 then illustrates the use of the language, with a toy example from the mobile robot domain. In our development of the language, we learned many lessons that are not covered in the purer AO language definitions; before concluding, Section 7 discusses a number of these lessons.

2 Related Work

A number of deliberative agent languages and architectures are already present in the literature. While some of these explore agent language theories, others have focused on the provision of complete agent frameworks.

Inspried by Dennett's Intentional Stance [10], Shoham's AgentO [1] established AOP as a programming paradigm on a par with OOP. Importantly, AgentO provided a clean language that could allow designers to build programs from mental attributes such as Beliefs and Commitments. Limitations of AgentO included no explicit support for Declarative Goals, a lack of plan operators, and little detail of how the high level programming language could interact with lower level code.

Rao's AgentSpeak(L) [3], based on the Procedural Reasoning System (PRS) [11], offered a more powerful agent design; AgentSpeak(L) agents had a mental state including Beliefs, Plans, Goals, Actions, Events, and Intentions. Rao's language is notable first as an AOP language grounded in action, and second as an AOP language that can be formally verified. However, the language intentionally left open the question of what choice functions should be used in the selection of intentions and plans. This, along with a limited set of plan operators, meant that AgentSpeak(L) was not, by itself, a suitable candidate for a deployable AO language.

A more recent AO language worthy of mention is *Goal Directed 3APL* [12]. That language, like its more abstract parent 3APL [13], strives for a separation of mental attributes and the reasoning process. This separation, facilitated with meta-language programming, leads to a well defined programming language, with a clear Operation Semantics provided by Transition Systems. 3APL provides a rich set of mental objects including Beliefs, Goals, Plans, and various Reasoning Rules. However, 3APL does not provide a fixed deliberation process. Instead, it is intended that designers be allowed to directly specify an agent's deliberation process, as well as its capabilites and base knowledge. Also, as with AgentO, there is only a primitive relationship between a 3APL action and the manipulation or sensing of an agent's environment.

Other recent entrants into the deliberative agent development literature include Nuin [14] and Jadex [15]. Both of these are agent architectures that aim to provide a practical framework for BDI style agents. Nuin's underlying BDI model builds on AgentSpeak(L), addressing issues like AgentSpeak(L)'s lack of choice operators. However, like PRS and AgentSpeak(L), Nuin requires designers to define fitness functions for the selection of current attention (i.e. `select-focus`). The Jadex BDI model is more basic, including Beliefs, Plans, and *to-do* Goals (Commitments). Jadex lacks declarative goals and means-end reasoning, plus there is an unclear relationship between object oriented and agent oriented design in plan definitions. Both Nuin and Jadex directly address *middleware* concerns such as message transportation, migration, and yellow pages support. Although provision of such services is doubtlessly valuable, combining these *middleware* issues directly with agent language or architecture design potentially precludes the development of more powerful control algorithms.

We conclude that the literature includes rich, theoretical, languages with well defined semantics (3APL), along with BDI frameworks for the development of AO applications (Nuin). However, these approaches either lack explicit deliberative process description, or they are unclear in how the language would work in real world deployment. It is our belief, that for AOP to become a mainstream programming methodology, AO languages must: (1) utilise a rich BDI style feature-set; (2) have a clear deliberation process; and (3) be a core language upon which other *middleware* and application specific aspects of agent deployment can be built. In the rest of this paper we present AF-APL as a language that has matured through application experience. In doing so, we hope that some of some of our lessons learned can be valuable to others.

3 AF-APL Constructs

The main constructs of any Agent Oriented Programming Language inevitably centre around objects such as Beliefs, Goals, Commitments and Plans. We now overview AF-APL's collection of base objects and meta constructs.

3.1 Core Logical Constructs

From the object perspective, an AF-APL agent is defined as a tuple of mental objects:

$$Agent = \{\mathcal{B}, \mathcal{G}, \mathcal{C}, \mathcal{A}, \mathcal{P}, \mathcal{BR}, \mathcal{RR}, \mathcal{CR}, \mathcal{PR}\}$$

where $\mathcal{B}$ is the agent's Belief Set, $\mathcal{G}$ is the agent's Goal Set, $\mathcal{C}$ is the agent's Commitment Set, $\mathcal{A}$ is the agent's Actuator Set, $\mathcal{P}$ is the agent's Perceptor Set, $\mathcal{BR}$ is the agent's Belief Rule Set, $\mathcal{RR}$ is the agent's Reactive Rule Set, $\mathcal{CR}$ is the agent's Commitment Rule Set, and $\mathcal{PR}$ is the agent's set of Plan Rules, or simply Plans. We now informally introduce each of the AF-APL constructs that are used to create and manipulate these objects.

Beliefs. In AF-APL, a belief is represented with the BELIEF construct. By default a belief only persists for one iteration of the agent's execution cycle. This behaviour can be changed through the use of the NEXT, UNTIL, or ALWAYS constructs as required. To illustrate, we specify that an agent believes that Rem always likes beer, as follows:

```
ALWAYS(BELIEF(likes(Rem, beer)))
```

Perceptors. An AF-APL agent contain a non-empty set of perceptors $\mathcal{P}$, each of which enable the agent to acquire beliefs about its environment. The PERCEPTOR construct is used to declare a perceptor with a particular identifier and external code implementation. Each perceptor is fired in its own thread once per agent execution cycle, and any resultant beliefs are then added to the agent's belief set $\mathcal{B}$. For example, to define a Java based perceptor for receiving FIPA messages, one might use this construction:

```
BEGIN_PERCEPTOR
 IDENTIFIER ie.ucd.af.fipa.fipaReceive;
 CODE ie.ucd.af.fipa.Receiver.class;
END_PERCEPTOR
```

Here, the IDENTIFIER of the perceptor is a unique perceptor identifier within the AF-APL namespace. CODE defines a piece of *external code* that is used to implement the perception task. Perceptor implementation can be provided as Java classes or C libraries that make use of a defined Perceptor interface. Using these interfaces, Perceptors add new beliefs to the agent's belief set. Multiple CODE declarations can be defined, but only one piece of code is used on any platform. This can be useful when dealing with migrating agents over multiple hardware platforms. Take for example a mobile agent which crosses between a high performance PC, and a computationally limited handheld device. Conceivably, when the agent moves to the low-spec platform, it may need to switch to a more basic form of perceptor.

Actuators. AF-APL agents contain a non-empty set of actuators $\mathcal{A}$, each of which constitutes the most basic action an agent can perform and reason about. As with perceptors, actuators are defined in terms of a particular identifier, with an implementation provided in an external programming language. In addition, actuators have pre and post conditions defined; these conditions determining what must be true for an actuator to be invoked, and what should be true when an actuator has completed. Axioms of the language also state that if an actuator succeeds or fails, then the agent directly adopts a belief to that effect; this technique has been found to be extremely useful in allowing an agent to reason about the success of its own actions. To avoid locking of the agent execution cycle, actuators are fired asynchronously. An actuator for sending FIPA messages might be defined as follows:

```
BEGIN_ACTUATOR
 IDENTIFIER <BEHAVIOURAL>
   ie.ucd.af.fipa.fipaSend(?fipa_msg);
 PRE BELIEF(TRUE);
 POST BELIEF(send_success(?fipa_msg))
    | BELIEF(send_failure(?fipa_msg));
 CODE ie.ucd.af.fipa.Sender.class;
END_ACTUATOR
```

The actuator identifier (like the perceptor and plan identifiers) uses a namespacing convention which is similar to (and compatible with) the Java namespace. The actuator identifier normally only refers to the shortened form of this term (i.e. fipaSend(?fipa_msg), but the long form can be used to distinguish between two actuators with the same short form identifiers. Actuators are by default assumed to be functional in nature (expected to self terminate); however, this interpretation can be overwritten by declaring the actuator to be behavioural by using the optional BEHAVIOUR keyword in the IDENTIFIER (See the DO_BEHAVIOUR_UNTIL construct for more details).

Commitments. A commitment is a promise made by an AF-APL agent to attempt an action. In other words, a commitment is the mental equivalent of a contract. As such, it specifies the course of action that the agent has agreed to; to whom this agreement has been made; when it must be fulfilled; and under what conditions the agreed course of action becomes invalid (i.e. under what conditions the contract can be breached). At any time the agent can contain any number of commitments in its commitment set $\mathcal{C}$. For example, we can represent that Rem has committed himself to eat biscuits at 11am (as long as he believes that he has no lunch plans) with the COMMIT construct as follows:

```
COMMIT(Rem,
      11:00,
      !BELIEF(has(Rem,LunchPlans)),
      eat(biscuits))
```

The first argument taken by a commitment construct is the name of the Agent to whom this commitment has been made; this can be any literal term, or the keyword ?Self, which is replaced at runtime with the name of the AF-APL agent. The second argument is the earliest time at which the agent should attempt to realize the commitment. The start time may be specified absolutely in the standard international date format YYYY/MM/DD-hh:mm:ss, or relative to the adoption time of the commitment by affixing a '+' character to the start of a time (e.g. +01:00:00 means that the action should be started one hour after the commitment was adopted. The key term '?Now' can also be used to specify that the minimum start time for the action equals the commitment adoption time. The third term is the commitment's maintenence condition; if at any time after adopting the commitment the term is no longer entailed by the agent's belief set, then the commitment is dropped immediately. The final term is the action to be performed by the agent; The commitment holds a special place in the AF-APL language, since it is through commitment management, and commitment revision that an agent performs intentional action. Section 4.1 overviews of the commitment management process.

Goals. A goal is a state of the world - or set of beliefs - that an agent wishes to bring about. For example, the goal of causing the door to be closed is represented as:

```
GOAL(door(closed))
```

Agents may adopt goals using either the ADOPT_GOAL or ACHIEVE_GOAL plan operators, as described in section 3.1. Once adopted into $\mathcal{G}$, an agent will use means-end reasoning to attempt to determine a plan that can achieve the goal. If such a plan can be determined, a secondary commitment to that plan structure will form as a child of the original goal; fulfilling the plan then causes the original commitment to be dropped. To guarantee continued reactive behaviour of the agent, the means-end reasoning process is performed asynchronously - this behaviour is discussed further in section 4.

Belief Rules. At any time the agent has a number of Belief Rules that allow the agent to infer new beliefs from already existing beliefs. Therefore, the agent can decide if a friend is allowed to drink beer with the following Belief Rule:

```
BELIEF(friend_age(?friend,?years))
& BELIEF(greaterThan(?years,18))
 => BELIEF(canDrinkAlcohol(?friend));
```

The left hand side of a belief rule is referred to as a belief query, and is a conjunction of positive or negative current beliefs containing free or bound variables.

Belief queries can reference the agent's belief set directly, or make use of three relational operations equals(?x,?y), greaterThan(?x,?y) and lessThan(?x,?y).

Commitment Rules. As well as belief rules, an AF-APL agent holds a set of commitment rules, $\mathcal{CR}$, that allow the agent to rationally decide on a course of action based on its mental state. Take for example an agent who is hungry and has a piece of fruit. Under these circumstances, a rational action for the agent to take, would be to eat the fruit. We can specify this as a commitment rule in AF-APL with the following:

```
BELIEF(hungry(?Self)) & BELIEF(haveFruite(?Self,?fruit))
  => COMMIT(?Self,
            ?now,
            BELIEF(TRUE),
            eat(?fruit));
```

The left hand side of a Commitment Rule is a Mental State Query. As well as allowing querying of $\mathcal{B}$, a mental state query can also query $\mathcal{G}$ and $\mathcal{C}$. Thus, an agent can adopt commitments based not only on its beliefs, but also on the actions it has already committed to. It is often useful to make judgements based on what the agent believes to be true in terms of commitment precedence and timing. We therefore introduced two mental functions which allow an agent to determine basic relationships between commitments; *before* allows an agent to determine if the initial start time of one commitment precedes another; while *consequenceOf* allows an agent to explicitly determine if one commitment is a direct result of another commitment (e.g. is a descendent).

Reactive Rules. An AF-APL agent contains a set of Reactive Rules $\mathcal{RR}$ that allow an agent to invoke an action directly based on what it believes to be true. The reactive rule is simpler than a commitment rule, but is less robust and its consequences cannot be reasoned about by the agent. A reactive rule can be used to code basic reactive behaviours into the agent's design. For example, a reactive rule to dodge an obstacle blocking a robot's progress could be encoded with:

```
BELIEF(blocked(ahead)) & BELIEF(moving(forward))
    => EXEC(dodgeObstacle);
```

Reactive rules are different from their big brothers - the commitment rule - in a number of ways, including: static binding, reactive rules are bound at runtime to a specific plan implementation, whereas choices between plans and actuators can be made by commitment management at runtime; no deliberation, the agent cannot reason about reactive responses, nor can the reactive rule trigger contain COMMIT or GOAL structures; priority execution, reactive rules take precedence over commitment rules; more basic formulation, reactive rules may only contain actuator identifiers or the SEQ, XOR, OR, PAR, and AND plan constructs.

Plan Operators. In all the commitment rule and reactive rule examples above, only single actions were to be performed. As would be expected, we explicitly provide a number of plan operators for constructions of complex action from primitives.

- **SEQ -** The Sequence Construct defines a set of steps that must be realized in the order listed. For example, the actions required to boil a kettle can be combined as:

  ```
  SEQ(getKettle, fillKettle, boilKettle)
  ```

  SEQ, along with PAR, AND, OR, and XOR are collectively known as the arrangements operators. These operators can operate on any whole number of arguments. The case of operating on one argument is not particularly interesting; the operator is said to succeed if its one argument succeeds, and fails otherwise.
- **PAR -** The Parallel Construct defines a set of steps that should be realized simultaneously. We can express the parallel actions of stirring and adding milk as:

  ```
  PAR(stir,addMilk)
  ```

- **AND -** The Random Order Construct defines a set of steps which can be realized in any order, parallel or sequential. All steps must be performed, but it really does not matter what order the steps are performed in. Coming back to the tea example: consider the actions of getting tea, getting a cup, and getting the milk. There is no particular ordering necessary here; therefore we can use the AND construct as follows:

  ```
  AND(getTea, getCup, getMilk)
  ```

- **OR -** The Non-Deterministic Choice Construct defines a set of steps, one of which must be realized. All steps are attempted in parallel. Once one of these steps is realized, the construct is said to have succeeded; resulting in all other steps being abandoned. If none of the steps return true, then the construct is said to have failed. Using the tea example again, it is of course possible to boil water in more than one way. In addition to using a kettle, water may also be boiled in a pot. This results in two alternative ways to get boiling water, which can be expressed in a plan body as:

  ```
  OR(SEQ(getKettle, fillKettle, boilKettle),
     SEQ(getPot, fillPot, boilPot)
     )
  ```

  The OR construct is useful when we wish to try out many different options at the same time. However, in practice, it is not always useful to perform all operations together. In our boiling water example, it is probably a bad idea to both try to get boiling water from the pot and from the kettle at the same time. Instead, we often try each option in turn; to do this we use the XOR construct.

– **XOR -** The Deterministic Choice Construct defines a set of steps, which must be performed in the order presented, and which succeeds when one of the steps is realized. Our boiling water example can be expressed with XOR:

```
XOR(SEQ(getKettle, fillKettle, boilKettle),
    SEQ(getPot, fillPot, boilPot)
   )
```

– **FOREACH -** The Universal Quantification Construct allows an agent to check the contents of its mental state, and assign variables into plans based on this mental state. FOREACH takes two arguments: a belief query sentence, and a plan body with free variables (all of which must be potentially scoped by the belief query sentence). To illustrate consider an agent that wants to hold a party, and therefore wishes to invite all its friends to the party; we can express this with the FOREACH construct as follows:

```
FOREACH(BELIEF(friend(?name)),AND(invite(?name)))
```

At runtime, if the agent's belief set includes the beliefs that it has friends: Anne, Jane and Freddy, then the above statement will be expanded to:

```
AND(invite(Anne), invite(Jane), invite(Freddy))
```

The plan body to be expanded must: (a) be based on an arrangement construct e.g. XOR, OR, AND, PAR or SEQ; (b) contain only one argument. If the belief query sentence fails - in this case, because the agent has no friends - then the FOREACH construct is said to fail.

– **TEST -** The Belief Query Construct allows us to test if particular beliefs are held by the agent. The construct takes one argument, a belief query sentence, which may or may not contain free variables. If the agent's mental state entails the belief query sentence, then the construct is said to succeed. We can use TEST to decide if we want to drink the tea that we have made:

```
SEQ(tasteTea,
 XOR(SEQ(TEST(BELIEF(tea_tastes(good))), enjoyTea),
     SEQ(TEST(BELIEF(tea_tastes(bad))), drinkItAnyway),
     drinkCoke))
```

Using the TEST construct is equivalent to creating an actuator which explicitly checks the mental state of the agent; however, the construct is defined as part of the language; meaning it is used more efficiently than an external actuator.

– **DO_BEHAVIOUR_UNTIL -** The Behaviour Controller Construct is used to handle actuators which have a behavioural rather than functional nature. All of the examples above presumed actuators to have a functional nature, in that they were expected to terminate eventually. Some actions, particularly in robotics, have something closer to a behavioural nature in that they do not have a natural termination point, and are only stopped once the agent believes something to be true. The DO_BEHAVIOUR_UNTIL construct takes two arguments: (a) the identifier of a actuator which has been declared BEHAVIOUR; (b) a belief query sentence, which can contain free variables. The actuator is invoked and will be left to run until it returns or the agent's belief set entails the query; in which case, the actuator is forcefully stopped. We can specify that a robot is to move along a wall until a door is found with:

```
DO_BEHAVIOUR_UNTIL(followWall, BELIEF(found(Door)))
```

– **TRY_RECOVERY -** The Error Recovery Construct is used to indicate whether an action - or indeed a plan structure - should have error recovery mechanisms associated. The construct takes two arguments: first, the action or plan body to be monitored for failure; second, the plan which is to be used for recovery. Unlike the *try catch* exception handling mechanisms in Java, TRY_RECOVER attempts to repair an erroneous situation and return the agent to finish the original action. For example, a robotic agent that has to perform a number of movement actions, can use a social recovery plan [16] to guard against erroneous situations as follows:

```
TRY_RECOVER(SEQ(moveForward("5m"),turn("90d"),enterDoor),
            social_recovery)
```

– **CHILD_COMMIT -** The Commitment Adjustment construct is used to override the default secondary commitment creation semantics of the language. As with the COMMIT construct introduced above, the CHILD_COMMIT construct takes four arguments, explicitly overriding to whom the secondary commitment is made; when the commitment is to be first attempted; under what conditions the commitment is dropped; and the action to be achieved. If only a subset of these parameters is to be overwritten, then the ?default key term can be used to indicate that the default value is to be kept. For example, if within a complete plan, there are two branches, one of which is to Jim, with another branch to Anna, then we can express this as follows:

```
CHILD_COMMIT(SEQ(COMMIT(Jim,?default,
                        ?default,doA),
                 COMMIT(Anna,?default,
                         ?defualt,doB)))
```

This construct is discussed further in the context of commitment management in section 4.1.

– **ADOPT_GOAL -** The Goal Adoption Construct is a mental action which adds a goal to the agent's goal set, returning immediately. Let us consider a robot which has been requested to close the door. The agent can adopt the general goal of causing the door to be closed with the following commitment rule:

```
ADOPT_GOAL(GOAL(closed(door)));
```

– **ACHIEVE_GOAL -** The Goal Achievement Construct adds the goal to the agent's mental state, but does not return until the GOAL is achieved.

Plans. All of the plan operators above allow a plan to be constructed out of pre-defined actions. We refer to such constructions as *Plan Bodies.* To facilitate code re-use, plan bodies can be wrapped in plan constructs. The plan construct is similar to the actuator construct, but takes a plan body as activity, rather than a piece of external application code. The plan operators introduced above can operate on these plans as well as actuators, thus allowing the recursive definition of plans. From our tea making scenario, a (naive) plan to make a cup of tea can be represented as follows:

```
BEGIN_PLAN
 IDENTIFIER ie.ucd.assitant.makeTea();
 PRE BELIEF(TRUE);
 POST BELIEF(made_tea);
 BODY SEQ(AND(getTea,getCup,getMilk),
          XOR(SEQ(getKettle, fillKettle, boilKettle),
              SEQ(getPot, fillPot, boilPot)),
          AND(addTea,pourWater),
          PAR(stir,addMilk),
          tasteTea);
END_PLAN
```

3.2 Meta Constructs

In addition to the base language concepts discussed above, we have found that a number of extra constructs - that do not strictly from part of the underlying logical language - can vastly improve the usefulness of AF-APL. We now look at two of these *meta constructs.*

Modules. An AF-APL module provides a simple way of declaring an external block of code (Java class or C object/library) which can be used to share functionality and memory between different actuators and perceptors. The construct takes one argument, the name of the module to be loaded by an agent. For example, a Java library which implements a hash-map, might be declared with the following:

```
MODULE myProject.mods.MyHashMap.class;
```

The module is instantiated and is agent specific, meaning all actuators and perceptors within an agent can have direct access to a common data source, which can be convenient when these actuators and perceptors need to share data, which is large in volume and not suited to being reasoned about by the agent.

Role Classes. AF-APL provides a form of inheritance in agent design through the use of explicitly defined role classes. AF-APL Roles allow a collection of actuators, perceptors and other agent components to be grouped together into an agent prototype. These agent prototypes can then either be instantiated directly into agents, or included in other agent designs. For example, the AgentFactory runtime [17] provides a number of pre-defined roles, including one that implements basic functionality for FIPA compliant communication. This role can be included into a new agent design with the USE_ROLE construct:

```
USE_ROLE ie.ucd.core.fipa.role.FIPARole;
```

Macro Inclusions. To improve code reuse capabilites, Plans, Actuators, and Perceptors can all be coded directly in their own files, and later included into an agent with the AF-APL meta constructs PLAN, ACTUATOR, and PERCEPTOR. For example, we can include the actuator for sending FIPA compliant messages as follows:

```
ACTUATOR ie.ucd.af.fipa.fipaSend(?fipa_msg);
```

4 The AF-APL Execution Model

The AF-APL programming constructs, overviewed in the last section, must be related to each other and interpreted at runtime. We now discuss our execution model and commitment management system that is used to rationally drive an AF-APL agent. The agent's Life Cycle is built out a of number of Execution Cycles. AF-APL provides a well defined (but extensible) execution model which reflects the basic needs of a hybrid agent. Unlike traditional Sense - Plan - Act approaches, the model is asynchronous, guaranteeing the continued operation of an agent - regardless of potential locking of 3rd party actuator and perceptor code.

The AF-APL execution cycle is presented in pseudo-code in figure 1. The first phase of the execution cycle concerns the update of the agent's belief set, $\mathcal{B}$, based on the results of perception and temporal belief update. To guarantee that the agent's execution cycle is not perturbed by potentially poorly designed 3rd party code, the perceptor firing and reading mechanisms are asynchronous. Actual perceptor code (e.g. C libraries) run outside of the agent's thread of execution; Instead, an interface to a perceptor is provided that includes a Belief Queue and an Event Queue. At this first phase of the execution cycle, each perceptor's Belief Queue is emptied into $\mathcal{B}$; followed by the sending of a *triggered* event to advise the native perceptor code that it is time to re-fill its Belief Queue. Handling of the perceptors is then followed by temporal belief update. For each of the agent's temporal beliefs, the agent's beliefs are updated in accordance with the semantics of the temporal operators used.

The second phase of the execution cycle then follows (line 7), causing reactive actions to be run in their own threads, if the reactive rule condition is entailed by the $\mathcal{B}$.

The invocation of reactive rules is immediately followed by the deliberation phase. Our design of this phase was influenced by our wish to provide our agents

```
FOR_EACH $p \in \mathcal{P}$ {
   $\mathcal{B}$ < p.queue;
   trigger(p);
}FOR_EACH $b \in \mathcal{BR}$ {
   IF b.condition {add(b,$\mathcal{B}$);}
}FOR_EACH $r \in \mathcal{RR}$ {
   IF r.condition {trigger.action;}
}FOR_EACH $g \in \mathcal{G}$ {
   IF g.state == achieved {drop(g);
   }ELSE_IF g.state == new {deliberate(g);
   }ELSE_IF g.state == plan_found {adopt(g.plan);}}
FOR_EACH $cr \in \mathcal{CR}$ {
   IF cr.condition {add(cr.Commit,$\mathcal{C}$);}
}FOR_EACH $c \in \mathcal{C}$ {
   process(c);
   IF c.state == fulfilled || c.state == redundant {
        drop(c);}
}
```

Fig. 1. AF-APL Execution Cycle

with goal directed reasoning, while guaranteeing the agent's continued reactivity. These requirements immediately rule out direct means-end reasoning in the agents execution thread. Two alternative approaches were investigated. The first saw the execution of a number of means-end reasoning steps per execution cycle, where the semantics of means-end reasoning were directly defined in AF-APL. The second saw the use of an external planning algorithm which runs in an asynchronous manner similar to actuator execution. Of these two choices, practical considerations directed us towards the later option, since it allows more freedom in the design and implementation of the planning algorithm. During this phase of the agent execution cycle, each of the agent's goals are analysed. If a goal has been achieved, then it, along with any subsequent commitments, are immediately dropped from the $\mathcal{G}$ and $\mathcal{C}$ respectively. If a goal is new, then means-end reasoning is triggered asynchronously; and, if a solution has been found through means-end reasoning, then a commitment to the resultant plan is adopted. A drawback to the use of an asynchronous planning model is that by the time the planning process has returned, the plan may be invalidated by changes in the real world or agent state. We argue that this is a problem that will be common to any agent acting in the real world, and is not specific to an asynchronous execution cycle.

Following deliberation, an AF-APL agent then addresses the questions of which commitments are to be adopted, and how these commitments are to be fulfilled. At a high level, this Commitment Management process involves the adoption of new primary commitments based on the values entailed by the agent's mental state; followed by the partial achievement of these commitments through the management of commitment structures, and execution of actuators. Our approach to commitment management is detailed in the following section.

4.1 A Model of Commitments and Commitment Management

In section 3.1 we briefly introduced our notion of a commitment as a promise that is made by an agent, to perform an action, for some agent (possibly itself), More precisely, an AF-APL commitment, $c \epsilon \mathcal{C}$, is a 6-tuple:

$$c = \{\alpha, \theta, \mu, \pi, \rho, \omega, \gamma\}$$

where α is the name of the agent to which the commitment has been made; θ is the earliest time at which the agent will attempt to fulfil the commitment; μ is the maintenence condition of the commitment; π is the activity that has been committed to; ω is an ordered set of child commitments; ρ is the state of the commitment; and γ is the commitment's parent (potentially null). For brevity, α and θ can be viewed from a common sense interpretation. μ is a belief sentence - or conjunction of beliefs - which must be obeyed by the agent for as long as the commitment is to be held by the agent; If at any point this belief is not entailed by $\mathcal{B}$, then the commitment is dropped.

π is the activity which has been committed to by the agent. In our model, π is a placeholder for an actuator, plan, or plan operator; has its own state; and can generate a number of child activities. The semantics of an activity are directly dependent on the semantics for the plan operator, plan, or actuator object as appropriate. The states of an activity are: NEW, an activity has been initialised; ACTIVE, the actuator, plan, or plan operator is active; SUCCEEDED, the activity has succeeded; FAILED, the activity has failed.

The state of a commitment ρ signifies at what stage of fulfilment the commitment is at. The commitment's state is important both from a conceptual view, in that it represents whether we are actively achieving a commitment, or if the commitment is planned to be achieved at a later time. It is also used in the commitment management model presented in the next section. A commitment can have five non-trivial states: DEPENDENT, a commitment that has been adopted, but is not due to be actively fulfilled until some fruition constraint has been achieved; PENDING, a commitment that the agent will begin to actively fulfil by the agent in its next execution cycle; ACTIVE, a commitment that is being actively pursued by an agent; REDUNDANT, a commitment that has failed - the maintenence condition for the commitment no longer holds; FULFILLED, a commitment that has been fulfilled - the activity committed to has succeeded.

Each commitment can have a number of child commitments' ω, where each child is a commitment to some activity, which if achieved, can partially achieve some part of the parent commitment. For example, a commitment to perform two actions, can be refined to two commitments to more primitive actions. Thus, a commitment structure can be formed, where a coarse grained commitment can be broken down into sub-structures of finer grained commitments. We call the topmost commitment in a commitment structure a primary commitment. The agent's commitment set $\mathcal{C}$ contains a number of these commitment structures, where each structure was added through the initial adoption of a primary commitment through the application of commitment rules or deliberation.

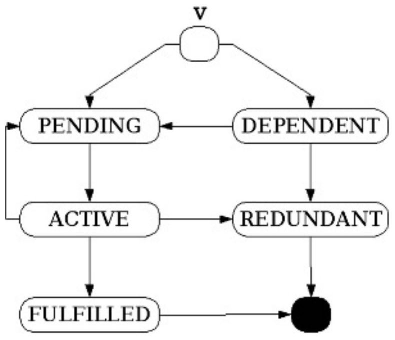


Fig. 2. Commitment State Diagram

Commitment Management. Above, we gave a description of an AF-APL commitment and showed the relationship between primary commitments, activities, and commitment structures. In this section, we look at AF-APL's commitment management algorithm. Once per execution cycle, each of the agent's commitment structures are refined, which involves the pruning and expansion of the structure. A pseudo-code representation of our commitment management algorithm is presented in figure 3.

During the first refinement phase, the algorithm descends to the leaves of the commitment structure, and examines the state of each activity committed to. If a commitment's activity has succeeded, then the commitment's state is set to FULFILLED. Else, if the activity has failed, the commitment's maintenence condition is checked. If this is no longer entailed by $\mathcal{B}$, then the commitment enters the REDUNDANT state; otherwise, the activity is reset. If the activity is still active, the maintenence condition is similarly checked - failure of the maintenence condition results in the commitment becoming REDUNDANT. By default, the leaf commitments, like all secondary commitments are created with a *weak minded* maintenence condition, meaning that if the committed to action fails, then the commitment enters the REDUNDANT state. The checking of maintenence condition, and notification of parent commitment's activities peculate up through the structure until all commitments have been checked. As parent commitments are notified of the change of state of child commitments, other previously DEPENDENT child commitments can become moved to the PENDING state. Such transitions are dependent on the semantics of the activity committed to. It should be noted that since AF-APL uses an open-minded commitment model, a commitment fails if the commitment's maintenence condition no longer holds - not if the activity committed to fails. Thus, if a commitment's activity fails, but the maintenence condition still holds, then the commitment is moved back to the PENDING state.

During the second refinement phase, all PENDING commitments are moved to the ACTIVE state, with a subsequent transition of the commitment's activity from the NEW to the ACTIVE state. The consequence of moving the commitment's activity to the ACTIVE state, is to create any new secondary

```
process_commitment(c){
  prune(c);
  expand(c);
}

prune(c){
  IF c.$\omega$.size == 0{
     IF c.$\pi$.state == SUCCEEDED{
         c.$\rho$ = FULFILLED;}
     ELSE IF c.$\pi$.state == FAILED{
        IF c.$\mu$ THEN c.$\rho$ = PENDING;
        ELSE c.$\rho$ = REDUNDANT;
   }ELSE{
        IF !c.$\mu$ THEN c.$\rho$ = REDUNDANT;}}
   ELSE{
      FOR_EACH $child \in c.\omega${
        prune(child);
        IF c.$\pi$.state == UPDATED{
           c.$\omega$.update();}
        ELSE IF c.$\pi$.state == SUCCEEDED{
           c.$\rho$ = FULFILLED;}
        ELSE IF c.$\pi$.state == FAILED{
           IF c.$\mu$ THEN c.$\rho$ = PENDING;
           ELSE c.$\rho$ = REDUNDANT;}}}
}

expand(c){
  IF c.$\rho$ == ACTIVE{
    FOR_EACH $child \in c.\omega${
       expand(child);}}
  ELSE IF c.$\rho$ == PENDING{
      IF c.$\theta$ {
         Set c.$\rho$ = ACTIVE;
         Set c.$\pi$.state = ACTIVE;
         FOR_EACH activity $\in$ c.$\pi$.$\omega${
           c_new = new c(activity);
           add(c.$\omega$,c_new);}}}
 }
```

Fig. 3. AF-APL Commitment Management Algorithm

commitments, marking these new commitments as DEPENDENT, ACTIVE, or PENDING as appropriate. The exact actions that are performed by during the setting of an activity to ACTIVE are dependent on the semantics of AF-APL's planning operators, plan and actuator constructs. A detailing of these semantics is considerably beyond the scope of this paper; but in summary, if the activity resolves to an actuator, then activation involves the triggering of the actuator in it's own thread; If the activity corresponds to a plan identifier, then a child

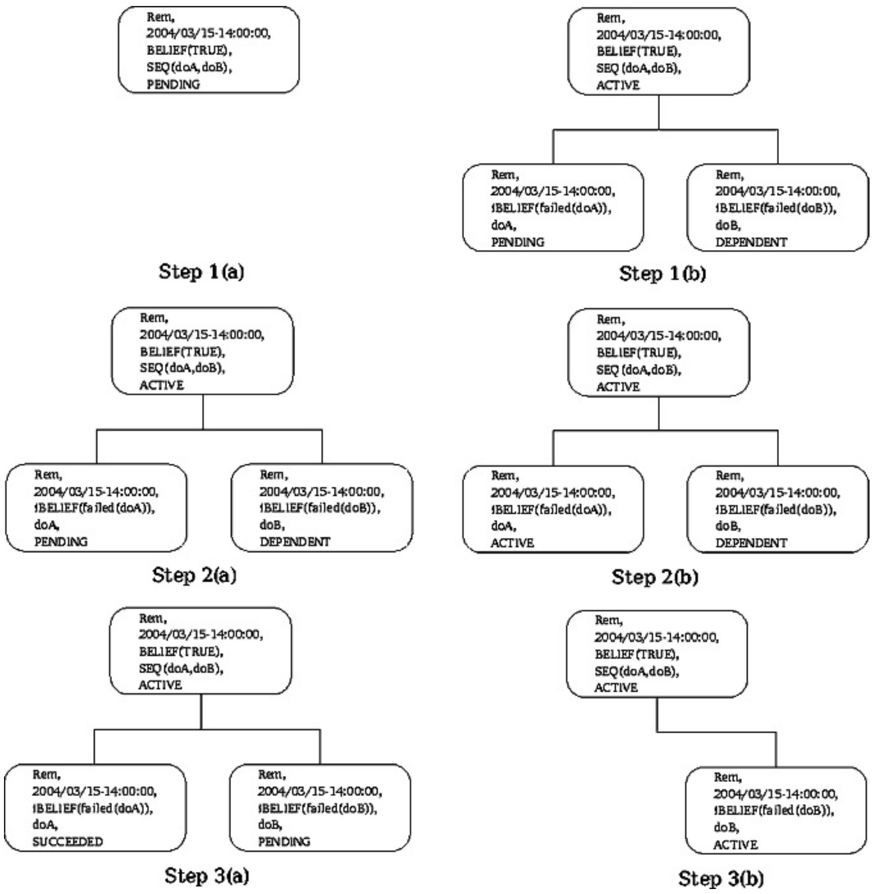

Rem, 2004/03/15-14:00:00, BELIEF(TRUE), SEQ(doA,doB), PENDING
Step 1(a)
Rem, 2004/03/15-14:00:00, BELIEF(TRUE), SEQ(doA,doB), ACTIVE
Rem, 2004/03/15-14:00:00, !BELIEF(failed(doA)), doA, PENDING
Rem, 2004/03/15-14:00:00, !BELIEF(failed(doB)), doB, DEPENDENT
Step 1(b)
Rem, 2004/03/15-14:00:00, BELIEF(TRUE), SEQ(doA,doB), ACTIVE
Rem, 2004/03/15-14:00:00, !BELIEF(failed(doA)), doA, PENDING
Rem, 2004/03/15-14:00:00, !BELIEF(failed(doB)), doB, DEPENDENT
Step 2(a)
Rem, 2004/03/15-14:00:00, BELIEF(TRUE), SEQ(doA,doB), ACTIVE
Rem, 2004/03/15-14:00:00, !BELIEF(failed(doA)), doA, ACTIVE
Rem, 2004/03/15-14:00:00, !BELIEF(failed(doB)), doB, DEPENDENT
Step 2(b)
Rem, 2004/03/15-14:00:00, BELIEF(TRUE), SEQ(doA,doB), ACTIVE
Rem, 2004/03/15-14:00:00, !BELIEF(failed(doA)), doA, SUCCEEDED
Rem, 2004/03/15-14:00:00, !BELIEF(failed(doB)), doB, PENDING
Step 3(a)
Rem, 2004/03/15-14:00:00, BELIEF(TRUE), SEQ(doA,doB), ACTIVE
Rem, 2004/03/15-14:00:00, !BELIEF(failed(doB)), doB, ACTIVE
Step 3(b)

Fig. 4. Commitment Structure Management

commitment to the plan body is adopted; If the activity corresponds to a plan operator (e.g. XOR, SEQ, TEST) then the creation of child commitments, is very much dependent on the semantics of the operator in question.

Although secondary commitment creation is, in general, dependent on the semantics of AF-APL's activity types, the values for a secondary commitment's agent α, earliest start time θ, and maintenence condition μ are specified by the basic commitment model; the default values for each secondary commitment's α and θ are directly inherited from their parents, while the default μ for each agent is that the activity being committed to has not failed. These default secondary commitment creation values can be overwritten through the use of the CHILD_COMMIT construct introduced in Section 3.1. Figure 4 shows the first three commitment management steps following the triggering of the following commitment rule by Rem:

```
BELIEF(request(doStuff)) => COMMIT(?Self,
                                   ?now,
                                   BELIEF(TRUE),
                                   SEQ(doA,doB));
```

In the first step, we see the initial adoption of the primary commitment to the SEQ(doA,doB), followed by the expansion of the primary commitment to a commitment structure. In the second step, doA is moved from the PENDING to the ACTIVE state - consequently triggering the external actuator code for doA (if we assume doA does in fact represent an actuator). In the third step, we assume that the doA actuator has successfully returned, thus causing the commitment to doA to be moved to the SUCCEEDED state, with the commitment to doB hence moved from the DEPENDENT to the PENDING state. If shown, this PENDING commitment would then be moved to ACTIVE in a subsequent commitment management step.

Our commitment management model is motivated by the dual constraints of allowing an agent to reason about its commitments, while guaranteeing the reactivity of the agent. For example, with our model, when a plan becomes active, then the commitment structure is fully expanded immediately, allowing an agent to reason directly about the commitments in the commitment structure.

5 Interpreting AF-APL: AgentFactory

In this section we introduce the AgentFactory Development Framework - along with its AF-APL compliant agent interpreter[1]. The focus of this paper is on the AF-APL language rather than any one interperter design. Therefore, this overview is brief and intended to elaborate on how an AF-APL interpreter might be implemented.

The AgentFactory Framework [4, 17] is a complete agent prototyping environment for the fabrication of deliberative agents. As shown in figure 5, AgentFactory can be conceptually split into two components: the AgentFactory Runtime Environment (AF-RTE) and the AgentFactory Development Environment (AF-DE). The AF-RTE includes an agent interperter; a library of standard actuators, perceptors, and plans; a platform management suite, which can hold a number of running agents at any time; and an optional graphical interface to view and manipulate agents and the platform. The AF-DE includes an agent/role compiler along with an Integrated Development Environment.

The Agent*Factory* interperter uses factory instantiation to allow more than just AF-APL agents to be run. This was achieved by designing the interpreter around a number of interfaces - where each interface manages one of the most common component types in agent designs. Figure 6 gives an abstraction of the interpreter design. In our design, an AF-RTE platform can have any number

[1] The AgentFactory Framework is available for download, along with full documentation on it and the AF-APL language, from http://www.agentfactory.com.

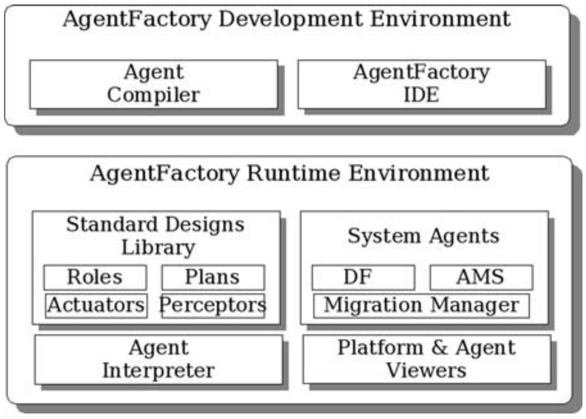


Fig. 5. The AgentFactory Framework

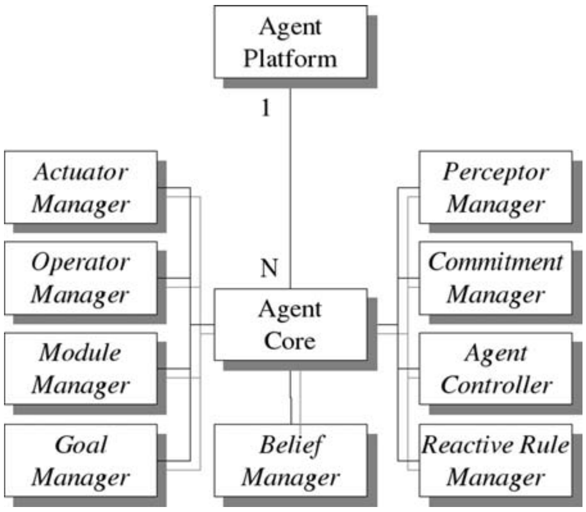


Fig. 6. AgentFactory Interpreter Design

of agents running - each agent is processed directly by an interperter instance that is accessed through an Agent Core object. During interpreter initialisation, the Agent Core class uses a platform configuration file to instantiate the appropriate objects for a given agent type, thus allowing agents designed in different languages to be instantiated on the same platform. In addition to providing abstractions of mental attribute types, the interfaces (typefaced in italics) also abstract the agent controller. Thus, although the interperter does not support scripting of agent controller designs, new controller variants can easily be implemented and tested.

AF-APL, as described in this paper, is only one of a number of AO language variants that can potentially be instantiated on the platform. We have found that employing a flexible interperter design is extremely beneficial for testing language variants, particularly when formal validation is not feasible.

6 Example Code

For illustration, we now present some code for a typical scenario that has influenced AF-APL's current features. Figure 7 presents a program fragment for an office assistant robotic agent. Starting from the top, we first see the use of the USE_ROLE macro to import AF-APL roles that provide domain specific definitions for FIPA compliant communication, and control of an autonomous wheelchair robot. The USE_ACTUATOR macro is then used to import a predefined actuator for vocalising utterances. This is followed by an in-line plan

```
USE_ROLE ie.ucd.core.fipa.role.FIPARole;
USE_ROLE de.tzi.RollandRole;
ACTUATOR de.tzi.MARY.speak(?utterance);

// Explicit Plan Definition
BEGIN_PLAN
 IDENTIFIER de.tzi.safeDeliver();
 PRE BELIEF(TRUE);
 POST BELIEF(delivered(parcel));
 BODY TRY_RECOVER(SEQ(DO_BEHAVIOUR_UNTIL(followWall,
                                          BELIEF(found(Door))),
                      enterDoor),
                  social_recovery),
END_PLAN

// Commitment Rule
BELIEF(requested(?agent,deliver(?object,?destination)))
& BELIEF(isSuperior(?agent)) & !COMMIT(?a,?b,?c,?d)
   => COMMIT(?agent,
             ?Now,
             BELIEF(TRUE),
             SEQ(fipaSend(?agent,
                          inform(ack(deliver(?object,?dest)))),
                 safeDeliver,
                 speak("I have a parcel for collection")
                 )
            );

// Reactive Rule
BELIEF(blocked(ahead)) & BELIEF(moving(forward))
   => EXEC(dodgeObstacle);
```

Fig. 7. Sample AF-APL Program

definition to deliver a parcel to a destination. The plan makes use of the TRY_RECOVER plan operator to initiate social help for recovery in the event that the basic delivery plan fails. The basic delivery plan also makes use of the DO_BEHAVIOUR_UNTIL operator which will cause the agent to follow the wall until it believes that a door has been found.

A commitment rule used states that if the agent believes that it has been asked to deliver a package, and if it believes that a superior made the request, and that the agent is not already committed to some other task, then the agent should immediately blindly commit to a simple plan. The plan uses an actuator inherited from the FIPARole to send an acknowledgement message to the requesting agent, before using deliver plan, to perform the delivery task. Upon arrived at the destination, the agent will announce its presence. In addition to the commitment rule, a reactive rule is also specified; this simple rule states that if the agent believes there is an object blocking its active path, then it will initiate a reactive action to dodge the obstacle.

7 Six Lessons Learned

Here, we list a number of lessons learned from our practical application of AOP - particularly in the robotics domain - and show how AF-APL has been formulated to benefit from these lessons.

- **Provide an Extensible Core Language.** An AO language should have a minimal conceptual footprint, while supporting extensibility based on domain specific requirements. AF-APL's actuator and perceptor interfaces act as membranes to domain specific code and abilities. This allows us to provide useful middleware features such as communication, yellow pages support, and migration, without the need to define these features within the semantics of the core language.
- **Provide a Practical Set of Planning Operators.** Real agents often need to perform a number of tasks in complex arrangements that go beyond parallel and sequential constructions. AF-APL's new plan operator set provides an improved, but modest, set of constructs to facilitate the construction of complex plans. We acknowledge that any set of operators is going to be found incomplete in some way, but AF-APL has been designed such that we can easily extend the semantics of the language through the addition of new or updated planning operators.
- **Support Reasoning on Goals & Commitments.** It is often extremely practical for an agent to reason about its own commitments, while deliberating over which new commitments should be adopted. AF-APL provides agents with these abilities through the well defined management of, and reasoning on commitment structures.
- **Support Robustness in Agent Operation.** Agent programs must guarantee the continued reactive behaviour of an agent - despite the possibility of bugs, crashes, or hanging in 3rd party actuator or perceptor code. All

aspects of AF-APL's execution cycle are asynchronous, insuring localised degradation in the face of 3rd party code.
- **Provide Multiple Levels of Reasoning.** Agent programming should provide a spectrum of control levels, ranging from full goal oriented deliberation through to reflective action. In addition to providing traditional Goal and Commitment directed deliberation, AF-APL allows for reflective behaviour within individual actuator or perceptor implementations; these are in addition to Reactive Rules that provide reactive abilities within the intentional layer.
- **Support Code Re-use.** For AOP to become useful in the real world, common software engineering practices will need to be supported by the languages. AF-APL provides code re-use facilities through the use of Role Inheritance and macro definitions and inclusions.

8 Conclusions and Future Work

We presented AF-APL as a language based on a formal agent treatment - but forged through lessons learned in application experiences. A key feature of the language is its asynchronous execution model that guarantees robust operation of the agent, even in the face of potentially poorly designed 3rd party code. Also, the language utilises a model of commitments that allow an agent to reason about its own future actions and goals in the pursuit of rational action.

AF-APL was originally given a formalisation around Possible Worlds Semantics. However, with the practically derived features presented here, we are currently in the process of re-formalising, based around an Operational Semantics. From a practical standpoint, we must further investigate the uses of roles and inheritance within AF-APL. We hope that such an effort will result in a greater understanding of how rapid prototyping, inheritance, and code-reuse can help to proliferate the AO paradigm.

Acknowledgements. We gratefully acknowledge the support of the Deutschen Forschungsgemeinschaft (DFG) through the SFB/TR 6023 project on Spatial Cognition - Subproject I3-SharC.

References

1. Shoham, Y.: Agent oriented programming. Artificial Intelligence **60** (1993) 51–92
2. Dastani, M., Dignum, F., Meyer, J.: 3APL: A Programming Language for Cognitive Agents. ERCIM News, European Research (2000) Consortium for Informatics and Mathematics, Special issue on Cognitive Systems, No. 53,.
3. Rao, A.: AgentSpeak(L): BDI Agents speak out in a logical computable language. In: proceeedings of the Seventh European Workshop on Modelling autonomous agents in a MultiAgent world, Institute for Perception Research, Eindhoven, The Netherlands (1996)

4. Collier, R.W.: Agent Factory: A Framework for the Engineering of Agent Oriented Applications. PhD thesis, University College Dublin (2001)
5. Bratman, M.: Intentions, Plans, and Practical Reason. Harvard University Press, Cambridge, MA, USA (1987)
6. Cohen, P., Levesque, H.: Intention is choice with commitment. Artificial Intelligence **42** (1990) 213–261
7. Georgeff, M., Lansky, A.: Reactive reasoning & planning. In: Proceedings of the Sixth Intenational Conference on Artificial Intelligence (AAAI-87), Seatle, WA, USA (1987) 677–682
8. Ross, R.J., O'Donoghue, R., O'Hare, G.: Improving speech recognition accuracy on a mobile robot platform using top-down visual cues. In: Proc. 18th International Joint Conference on Artificial Intelligence (IJCAI-03). (2003)
9. Phelan, D., Strahan, R., Collier, R., Muldoon, C., O'Hare, G.: Sos: Accomodation on the fly with access. In: Proceedings of the 13th International FLorida Artificial Intelligence Research Symposium Conference (FLAIRS 2004), Miami Beech, Florida (2004)
10. Dennett, D.C.: The Intentional Stance. The MIT Press, Massachusetts (1987)
11. Rao, A.S., Georgeff, M.P.: BDI agents: from theory to practice. In Lesser, V., ed.: Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95), San Francisco, CA, USA, The MIT Press: Cambridge, MA, USA (1995) 312–319
12. Hindrikis, K.V., de Boer, F., van der Hoek, W., Meyer, J.J.: Agent Programming in 3APL. In: In Proceedings of Autonomous Agents & Multiagent Systems (AAMAS) 1998. (1998)
13. Dastani, M., van Riemsdijk, B., Dignum, F., Meyer, J.J.: A Programming Language for Cognitive Agents: Goal Directed 3APL. In: Proceedings of AAMAS 03. (2003)
14. Dickinson, I., Wooldridge, M.: Towards practical reasoning agents for the semantic web. In: 2nd Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS-03), Melbourne, Australia (2003)
15. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: Implementing a bdi-infrastructure for jade agents. in: EXP - In Search of Innovation (Special Issue on JADE) **3** (2003) 76–85
16. Ross, R., Collier, R., O'Hare, G.: Demonstrating social error recovery with agentfactory. In: Proceeedings of The Third International Joint Conference on Autonomous Agents and Multi Agent Systems. (2004)
17. Collier, R.W., O'Hare, G., Lowen, T., Rooney, C.: Beyond prototyping in the factory of the agents. In: 3rd Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'03), Prague, Czech Republic (2003)

A Toolkit for the Realization of Constraint-Based Multiagent Systems

Federico Bergenti

Consorzio Nazionale Interuniversitario per le Telecomunicazioni
AOT Lab
`bergenti@ce.unipr.it`
`http://www.ce.unipr.it/people/bergenti`

Abstract. Autonomy is largely accepted as a major distinctive characteristic of agents with respect of other computation models. This is one of the main reasons why the agent community has been investigating from different perspectives constraints and the tight relationship between autonomy and constraints. In this paper, we take the software engineering standpoint and we exploit the results of the research on constraint programming to provide the developer with a set of tools for the realization of constraint-based multiagent systems. In detail, the purpose of this paper is twofold. In the first part it presents a model that regards multiagent systems in terms of constraint programming concepts. This model comprises an abstract picture of what a multiagent system is from the point of view of constraint programming and a language for modeling agents as solvers of constraint satisfaction and optimization problems. The second part of this paper describes an implemented toolkit that exploits this model to support the developer in programming and deploying constraint-based multiagent systems. This toolkit consists of a compiler and a runtime platform.

1 Introduction and Motivation

Autonomy is largely considered a characteristic feature of agents that differentiate them from other computation models [13]. Many researchers claim that autonomy is the one and only distinctive features of agents [3] and the large amount of work about goal-directed behavior [2] or, more generally, about rationality [9] has the ultimate goal of providing a scientific understanding of autonomy. Along these guidelines, the research that motivates the results described in this paper has been devoted to study the opposite side of the coin: if we accept that agents are inherently autonomous, then we need to face the engineering problem of constraining this autonomy in a reasonable way. Even if some researcher have a radically different opinion, see, e.g., [14], we believe that a major step we need to undertake to regard agents as applicable abstractions in the engineering of everyday software system is to find a reasonable way to constraint the autonomy of agents.

Rationality can be regarded as one way to achieve this purpose, but facts have already proven extensively that the use of rationality in real system is still remote. Nevertheless, we need some way to guarantee (at least) some very basic properties of engineered systems, e.g., safety and liveness, and we cannot simply accept that something, which is not under the control of the developer, might emerge and damage the system.

In this paper we consider a method for constraining the autonomy of agents that exploits the direct intervention of the developer in defining what are the boundaries of an acceptable behavior for an agent. This method relies on the results of the research on constraint programming because of two very basic reasons. The first is that constraint programming treats constraints, and therefore autonomy, as a first class metaphor. This allows the developer to manipulate constraints directly. Moreover it offers the great advantage, over other forms of management of the autonomy, of being independent of any grand principle, like rationality. Such principles are elegant and they permit to treat a large set of different problems homogeneously. Nevertheless, they often become subtle and difficult to manage for the developer because of their inherent problem-independence. Rather, constraint programming puts the focus on the problem at hand and it uses only the constraints that are embedded in the problem itself, and no other grand principle is required.

The second reason for choosing constraint programming for managing the autonomy of agents is that it is sufficiently powerful to allow the description of another very basic characteristic of agents: goal-directed behavior. The success of constraint programming in problems like scheduling [10] and planning [12] demonstrates that it can provide good results in supporting the desired goal-directed behavior of agents.

This paper presents the results of the work that we have been doing during last year, and that is based on the guidelines that we have just stated. The aim of this project is to deliver a set of constraint-based tools that everyday developers could adopt for the realization of their multiagent systems. We did not choose any reference application domain for such tools because we intended to provide an enabling technology capable of providing its benefits in many cases. Actually, we believe in the position that Freuder stated in its celebrated truism [4]: *"Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it."*

The description of our tools starts, in the following section, with an introduction to what constraint-based multiagent systems are for us. Then, in section 3 we present a programming language that exploits the model of constraint-based multiagent systems to provide the developer with a direct means for implementing agents as solvers of constraint satisfaction and optimization problems. This language is the core of the development toolkit that we present in section 4. This toolkit consists of a compiler and a runtime platform and it allows for a seamless integration of constraint-based multiagent systems with everyday applications. This work is summarized in section 5, where we outline some conclusions and we present some future direction of research.

2 Constraint-Based Multiagent Systems

The interplay between autonomy and constraints has been the subject of a large work in various research communities and under various points of view, e.g., classic works about this are [6, 7, 8]. In our work we take the point of view of software engineering and we see *agent-oriented constraint programming* as a particular approach to realize multiagent systems, rather than, e.g., a distributed approach to solve constraint satisfaction problems.

In order to give a more precise meaning to these words, and to show the advantages of this approach, we start from the foundations of constraint programming and we show how these can be used as guidelines for the realization of multiagent systems. It is worth noting that even if some objectives are in common, our approach differs significantly from concurrent constraint programming [11] because we decided to start from a more operational, and more modern, approach to constraint programming.

A *constraint satisfaction problem* (CSP) [1] is a tuple $< X, d, C >$, where X is a set of n variable names, $X = \{x_1, \ldots, x_n\}$, $d : X \to D \subseteq \mathbf{R}$ is a mapping between a variable name and the domain of its possible real values, and C is a collection of m constraints $C = \{C_1, \ldots, C_m\}$. Each constraint is a proposition over a subset of the available variable names $S \subseteq X$, called the *scheme* of the constraint.

In this definition we consider only real variables because all other sets we need to deal with, e.g., sets of strings enumerated extensively, can be mapped bi-directionally to subsets of real numbers.

A solution to a CSP is an assignment of values that maps all variable names in X with a value compatible with d that satisfies all constraints in C.

An interesting property of this definition of CSP is that it allows for an easy definition of sub-problem. This feature is of singular importance in our agent-oriented approach because we will exploit it in treating goal-delegation.

We say that CSP_1 is a *sub-problem* of a CSP_2 if $X_1 \subseteq X_2$, and all constraints in CSP_2 whose scheme is a subset of X_2 are in CSP_1.

For the sake of simplicity, in this paper we do not take into consideration the serious problem of possible mismatches between the representations of different problems. If two problems contain the same variable name, we assume that these names are two appearances of the same variable, which is then shared between the two problems.

In order to exploit these ideas in an agent-oriented fashion, we need to introduce some notion supporting the embodied nature of agents and their inherent goal-directed behavior. This is the reason why CSPs are not sufficient and we need to rely on a well-known extension of them.

A *constraint satisfaction and optimization problem* (CSOP) [1] is a CSP with an associated targets T, i.e., it is a tuple $< X, d, C, T >$ where $T : S \subseteq X \to \mathbf{R}$.

A solution of a CSOP is a solution of the underlying CSP that maximizes the target T.

We say that $CSOP_1$ is a sub-problem of $CSOP_2$ if the CSP associated with $CSOP_1$ is a sub-problem of the CSP associated with $CSOP_2$ and any solution of $CSOP_1$ is a partial solution of $CSOP_2$ for the set of variables of $CSOP_1$.

This definition of sub-problem relies on a very demanding condition and we need to alleviate it with the introduction of the notion of *weak sub-problem.* We say that a $CSOP_1$ is a weak sub-problem of $CSOP_2$ if the CSP associated with $CSOP_1$ is a sub-problem of the CSP associated with $CSOP_2$, and nothing is said on the two targets. This definition allows decomposing a problem into sub-problems and to solve the various sub-problems independently, without requiring the decomposition of the target of the original problem.

The approach that we propose to exploit these ideas for the realization of multiagent systems is to state that an agent is nothing but a solver of a particular CSOP and that each agent in a *constraint-based multiagent system* may have a different CSOP to solve.

Agents acquire from their sensors:

1. Constraints, that are inserted or removed from the set of constraints of their problems; and
2. Values for the variables of their problems, that can be asserted or retracted, i.e., particular constraints of the form $X = x$.

Agents act in accordance of the solution, even partial, they find for their problems. Once an agent has definitely found a partial solution to its problem, i.e., once it comes to know that it will not backtrack the partial assignment of its variables if nothing changes in its internal state, then it can act on the environment or it can perform communicative actions in the direction of other agents. The selection of the action to perform depends only on the partial solution it found.

The multiagent system as a whole is not associated with any CSOP, rather its behavior emerges from the interactions of all agents. Such interactions can occur through the environment, i.e., through shared variables, or though communication.

Communication between agents is represented in our model using the standard approach of communicative acts. But, differently from many other approaches, we largely reduce the problems of communicative acts because we take a minimalist approach to agent communication languages. We say that an agent communicates with another agent only to ask to this agent to perform some action, whether communicative or not, whose outcome would help it solving its problem. Basically, we say that an agent communicate with another agent only to delegate a weak sub-problem of its CSOP to it.

Exploiting this simple model of communication we have the advantage of decoupling the problem of actually choosing what to say and to who from the CSOP the agent solves. If we work under the assumption that all agents provide to a central repository the (true) list of problems they can solve, then an agent A_1 communicate with an agent A_2 only if:

1. The problem that A_2 solves is a weak sub-problem of the problem of A_1; and
2. A_1 cannot, or does not want, to solve this sub-problem on its own.

From a rather superficial point of view, this approach to communication seems too poor with respect of the standard approaches that one may expect from an agent model. This is only partially true, because this approach is a direct implementation of goal-delegation, which is largely considered the ultimate rationale of communication between agents. Moreover, this model is not limited to cooperative agents only, because agents are associated with different CSOPs and optimal solutions to such problems may be conflicting. In particular, the target of the CSOP of each agent can be assimilated to the utility function of that agent, and each agent has potentially a different target for its problem.

The model that we have just outlined is obviously ideal for a number of reasons. First, it does not include time as a first-class concept. Time can be modeled as a variable in the CSOP of an agent, but the tight coupling between such a variable and the resolution process, through the time spent for actually solving the problem, can cause severe problems. Moreover, our model allows actions only in result of a partial solution of the problem of an agent. This implies that an agent cannot perform actions for a part of the duration of the resolution process, which inherently depends on the problem the agent is facing. This is a classic inconvenience found in many other agent models and it raised crucial problems like belief and intention revision [5].

Such limitations of our model are significant, but for the moment we decided not to take them into consideration because we adopted the standard approximation of quasi-stationary environments. If the time needed to solve the problem is sufficiently small with respect of the expected time of reaction of the agent to a change in the environment, then the agent would behave as expected. We decided to adopt this assumption because our experience suggests that it is applicable in many realistic situations, especially if we target common software systems where, if it is sufficiently reactive for the user, then performances is not an issue.

Our model of agents as CSOP solvers has many resemblances with more traditional agent models, especially with rational agents. Anyway, we put a strong focus on the internals of the agent, rather than on its behavior seen from the outside, and therefore the ascription of a mental state to such agents seems a difficult problem. This difficulty justifies the assumption that our agents are not easily framed into other agent models and therefore we decided to find a new nickname for them capable of capturing their nature of atomic computational entities whose interaction animates the multiagent system as a whole. This is the reason why we use to refer to this particular sort of agents as *quarks*.

3 A Language for Agent-Oriented Constraint Programming

The model of quarks that we outlined in the previous section has a number of interesting properties. Among them, it is worth noting that it does not depend on any particular (and possibly implicit) restriction that a specific language for constraint programming might impose. Therefore, it offers a good level of generality and expressive power. Moreover, it is a good approach for studying the algorithms capable of controlling the behavior of the quark.

Nevertheless, we cannot simply give to the developer the notions of variable, domain and constraint. It is not a reasonable approach for a developer that is already familiar with the sophisticated modeling techniques that object orientation has promoted in the last twenty years. The basic problem is that the effort required for mapping a reasonably complex application domain into a set of variables, domains and constraints seems excessive. We definitely need to define a language supporting higher level abstractions and the rest of this section addresses this problem.

The *Quark Programming Language*, or QPL (pronounced *kju:pl*), was designed to provide the developer with a user-friendly approach to realizing quarks. We defined an abstract model of it and then we mapped this model to a concrete syntax. Exploiting this syntax we realized a compiler, with an associated runtime platform, for a concrete use of QPL in running systems. The compiler and the runtime platform are described in next section and what follows is an informal description of QPL. It is worth noting that the semantics of QPL has already been formalized with two different approaches. The first is an operational mapping between a QPL program and a CSOP. This is a crucial step because it allows mapping a quark written QPL with an algorithm for controlling its behavior. The second approach is based on a description logic and it is useful for understanding the expressive power of the language. For the sake of brevity, none of these semantics is presented here and the subject is left for a future paper.

The realization of a quark in QPL requires to define the class of quarks it belongs to. This class defines the common characteristics of each quark it comprises and it provides a means for the developer to instantiate quarks. A class of quarks is composed of:

1. A name;
2. An import section;
3. A public problem description section;
4. A private problem description section;
5. A targets section; and
6. An actions section.

The name of the class is used to support the creation of new quarks: the developer uses it when he/she wants to instantiate new quarks. The import section declares the external components that the quark will use. Such components are non-agentized, third-party software that a quark may need to exploit in its lifetime.

These two sections of the program of a quark provide a bi-directional link between the quark and the rest of the non-agentized software of the system. This is why they are (intentionally) described vaguely in the specification of QPL. Any possible implementation of QPL has to deal with its own means for instantiating quarks and with its interface with external components. In the implementation of QPL that the following section describes, quarks can be instantiated by means of an API available through a .NET and a WSDL interface. Similarly, external components are imported and accessed through a .NET and a WSDL interface.

After the import section, QPL requires the developer describing the problem that the quark will solve. Such a description deals only with the CSP part of the CSOP that the quark will solve. This description is split into a public section and a private section. The public section provides a description of the problem that is suitable for a publication in the central repository of the system. This description can be used to tell other quarks what this quark is capable of doing, i.e., what are the problems it can solve. This part of the description of the problem is used to support communication and goal delegation. The private section refines what the developer declared in the public section with an additional set of details that he/she needs to introduce to make the quark fully functional. Such details are not essential for other quarks to reason about the problem this quark solves, and these can be assimilated to implementation details that the good old principle of information hiding suggests to keep private.

Both the public and the private sections of the description of the problem are then split into the following sections:

1. A structure section; and
2. A constraints section.

The structure section describes the domain of the problem the quark will solve. This description is based on a classic process of classification that closely resembles the one we commonly use in object-oriented modeling. This section of a QPL program has the ultimate goal of defining a vocabulary for describing the constraints and for naming the variables of the CSOP of the quark. The constraints section uses the vocabulary identified in the structure section to define the constraints of the CSOP of the quark.

The structure section of a class of quarks is described in terms of:

1. A set of classes of objects of the domain of the problem;
2. A set of relations between such classes;
3. A set of catalog objects;
4. A set of enumerative types; and
5. A set of constrained predefined types.

A class of objects is composed of a set of attributes, each of which is described as a name and a type. The type of an attribute is one of the types that the developer defines when he/she declares its enumerative or constrained predefined types (points 4 and 5 in the previous list). An enumerative type is a set of elements expressed extensionally that belongs to one of the predefined types that QPL provides, i.e., string, double, integer, and boolean. All elements of this list belong to the same predefined type.

In cases where an extensive enumeration of values is not practical, the developer can define a subset of the values of a predefined type through a constrained predefined type. This is a predefined type plus a constraint that restricts its possible values. For the moment, QPL allows constraining only doubles and integers and it provides only three constraints: one for setting a minimum value, one for a maximum value and one for a step in the series of values.

A class of objects in QPL can be a:

1. Catalog class;
2. Configurable class; or
3. Abstract class.

A catalog class is a class of objects whose elements are extensively enumerated in the program of the quark. These enumerated values are called *catalog objects* and they are listed as point 3 of the features that the developer uses in the structure section of a class of quarks. In the concrete implementation of QPL that the following section describes, catalog objects can be enumerated directly in the program of a quark, or they can be imported from the tables of a relational database.

A configurable class is a class of objects whose elements are described intensionally. These classes are modeled only in terms of their attributes: each attribute has a set of valid values as it is associated with an enumerative type or with a constrained predefined type. The characteristic feature of these classes is that we do not provide any restriction on the values that various attributes might take in a single instance. The constraints that we will introduce later, in the constraint section of the class of quarks, will provide the conditions that the values of the attributes of a single instance of a configurable class must respect. This should make clear that the quark regards the attributes of configurable classes as variables of its CSOP. The ultimate goal of the quark is to assign a value to any attribute of any instance of any configurable class in its current solution.

The third type of class we can define in the structure section of a quark is that of abstract classes. An abstract class is a class that we use to collect a set of attributes common to a set of catalog classes into one single container. An abstract class is modeled only in terms of its attributes (just like a configurable class) but we need to subclass it with another abstract class, or with a catalog class, in order to give a meaning to it. Abstract classes are useful to provide an abstract view of (a superset of) a set of catalog classes. Moreover, they allow structuring the domain of the problem and expressing constraints on attributes shared by a set of catalog classes.

QPL supports the assembly of classes of objects though relations. Such relations between classes can be of three types:

1. Generalization/specialization, that expresses a superset/subset relation between the objects of two classes;
2. Association, that expresses a shared composition of the objects belonging to two different classes: each object of the container class is made of a number of objects of the contained class, and such objects can be shared among a number of relations; and
3. Containment, that expresses a private composition of the objects belonging to two different classes: each object of the container class is made of a number

of objects of the contained class, and such objects cannot be shared among relations.

The latter two relations are qualified with a cardinality that models the number of objects of the contained class that take part of the relation.

The structure section of a QPL program provides all features we need in order to define the variables and the domains of the variables of the CSOP. Each attribute of any instance of a configurable class is a variable of the CSOP. The domains of such variables equal the domains of the corresponding attributes. Such domains are specified using enumerative types of constrained predefined types.

In order to complete the description of the CSOP of the quark, we need to declare a set of constraints and a target. QPL provides two ways for describing a constraint:

1. Compatibility/incompatibility tables; and
2. Rules.

A compatibility/incompatibility table is a list of tuples that enumerates the possible values of a group of attributes. If the list contains permitted tuples, we talk of a compatibility table, otherwise we talk of an incompatibility table. The two representations are identical and the choice depends only on the number of entries in the two possible lists.

Compatibility/incompatibility tables offer an extensional means for modeling a constraint. On the other hand, rules provide an intensional form of describing a constraint. A rule is a proposition that must hold for any possible solution to the CSOP of the agent. QPL provide only one type of rule: *if/then/else* clauses. The building blocks that the developer can use to compose the three expressions of an if/then/else clause are:

1. The set of standard operators over integers, doubles, booleans and strings; and
2. A set of *attribute terms* that are found in the vocabulary defined in the structure part.

The attribute terms allow the developer to identify variables that the quark will use to validate the rule. The general form of an attribute term in a rule is:

```
class[#id](.relation[#id])*.attribute
```

If we forget about `#id` for a moment, this form simply allows navigating classes and relations to reach an attribute from an initial class called `class`. From this class we can exploit association and composition `relation`s to reach other classes. Once we reached the class that comprises the attribute we are addressing, we identify the `attribute` through its name.

The use of `#id`s allows narrowing the number of variables that this pattern matches. If we do not use any `#id`, all instance of all classes met during the traversal from `class` to `attribute` are used in the rule. For example, `PC.type` addresses the value of the attribute `type` of all instances of the class `PC`. The use of

`#id` allows choosing a particular instance of a relation or of a class and restricting the application of the rule only to such instance, e.g., `PC#0.hardDrive#1.speed` matches the attribute speed of the second hard drive of the first `PC` only.

QPL allows using `#ids` in conjunction with two general purpose operators: `sum` and `product`. These operators have the standard meaning of repeated sums and products and they can contain expressions where `#id` is replaced with a variable term.

The grammar that QPL provides for rules is completed with two shortcuts for expressing common constraints:

1. `always`, meaning `if true then`; and
2. `never`, meaning `if true then not`.

As an example, the following is a very simple rule in QPL that states that if the type of PCs is games, then any hard drive in the needs to be at least 20GB.

```
if PC.type = ‘Games’ then HardDrive.size > 20
```

Each rule in QPL can be relaxable and relaxable rules are assigned a priority that drives the CSOP solver in deciding which rule to relax in order to find a solution. In the standard semantics of QPL, rules are relaxed only to find a solution and no rule is relaxed if a solution is already available, e.g., in the attempt to find a better solution.

The definition of the CSOP of a quark is completed with the definition of a target. This is done in QPL through a list of expressions that evaluate to a number. Each one of these expressions is assigned a priority and a direction, that can be maximize or minimize. The standard semantics is that the quark tries to maximize/minimize the target expression with a given priority only if all other target expressions with greater priority are already maximized/minimized.

The expressions used to indicate targets are composed with the same building blocks that QPL allows for the expressions of an if/then/else clause, with the sole restriction that they must evaluate to a number.

A quark can act on the outside world when it finds a partial solution to the problem it is managing. QPL allows the developer to specify actions in terms of a fire condition and a concrete act. The fire condition is an expression that evaluates to a boolean value and that exploits the vocabulary and the grammar used for rules and targets. When a fire condition holds, i.e., when a partial solution that verify a condition is found, the quark performs the associated act. Fire conditions are prioritized and the quark performs the act associated with the fire condition with the topmost priority that currently holds.

The concrete description of acts is kept out of the language specification because it relies on the concrete implementation of QPL and on how quarks are enabled to interact with non-agentized, third party software. In the implementation presented in the following section, acts can be invocations of methods of .NET components (previously imported), or they can be invocations of Web services (previously imported). In both cases, the partial solution found can be used to supply the arguments to the invocation.

The definition of actions closes the definition of a class of quarks in QPL. This definition allows enumerating all building blocks that the quark needs to know which problem to solve and what to do during the resolution process. Only two aspects seems missing: some means to allow quarks sensing the environment, and some other means to allow quarks communicating. The problem of sensing the environment is intentionally left out of the specification of the language because of the same reason we mentioned for concrete acts. Quarks does not sense the environment actively, they come to know of any change in the environment because of changes in their problem. How external software, e.g., the manager of a sensor, can actually achieve this is part of the concrete implementation of QPL. In the implementation presented in the following section, a QPL program is compiled to a .NET class or to a WSDL interface and both provide an API for pushing information directly in the problem of the quark.

The problem of communication is basically the same as the problem of sensing. As we briefly mentioned in the previous section, our model uses a minimalist approach to communication that allows hiding the process of information exchange from the developer. Each concrete implementation of QPL will have its own way to exchange weak sub-problems between quarks. In the implementation presented in the following section, the runtime platform provides the central repository for publicizing the capabilities of quarks, and it exploits .NET and WSDL interfaces for concretely exchanging messages in the multiagent system.

4 A Toolkit for Agent-Oriented Constraint Programming

In this section we introduce the *Quark Toolkit*, or QK (pronounced *kju:k*), a toolkit we realized to support the developer in implementing multiagent systems based on the ideas we described in the previous sections. QK is made of two parts:

1. A compiler, that compiles the QPL program of quarks to executable modules; and
2. A runtime platform, that provides all facilities we need to deploy a multiagent systems.

The compiler of QK is a command-line tool that takes a set of QPL programs, one file for each single class of quarks, and generates a set of .NET classes and a set of WSDL interfaces. These classes and interfaces are dual and classes implement the relative interfaces. This approach allows using quarks in two ways:

1. As .NET components exposing a fairly simple API for two-way communication with the rest of the .NET system;
2. As Web services that can be integrated in any system capable of exploiting their WSDL interface.

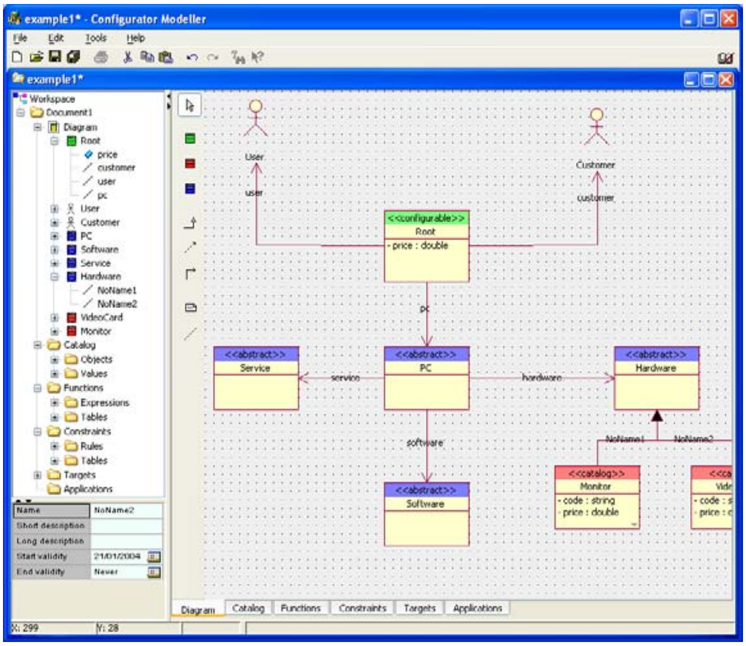
Both approaches are equally valid from the point of view of the developer and the pros and the cons of them have already been discussed largely after the introduction of .NET.

The QK compiler produces one .NET class for each single class of quarks. The interface of these classes does not depend on the problem a quark is designed to face, but it simply enables a bi-directional communication between the quark and an external .NET object. In particular, this interface provides a set of methods for informing the quark of new values for a variable or of new constraints in the problem. Then, it exposes a listener interface, together with a set of management methods, to allow an external component to observe the state of the reasoning process. Finally, this interface provides a few management methods that the QK runtime platform uses to manage the lifecycle of quarks and to interface a quark with the central repository of the platform.

The compiler of QK produces a .NET Intermediate Language (IL) source code, i.e., a source code of .NET IL mnemonics. This compilation is direct and it does not need to pass through a higher level language, e.g., C# or Java. The produced .NET IL is then translated into its executable form exploiting an IL assembler. The system has been tested with the two most popular IL assemblers:

```
agents ShopAssistant {
  uses webservice PriceManager = ‘http://...’;

  ...define what a PC is
  via composition, aggregation and inheritance...

  target minimize PC.delivery

  rule SOLO9100SE
    if PC.code = ‘SOLO9100SE’ then
      // Constraint on processor
      Processor.type  = ‘Pentium’        and
      Processor.clock => 300             and
      Processor.clock <= 366             and
      // Constraint on RAM memory
      RAM.type = ‘SO-DIMM’               and
      RAM.size >= 16 and RAM.size <= 384 and
      ...

  action EstimatePrice
    if PC.code = ‘SOLO9*’ then
      PC.price = PriceManager.
        EstimatePrice9XXX(PC, Customer);
}
```

Fig. 1. A simple QPL source code



Name	NoName2
Short description	
Long description	
Start validity	21/01/2004
End validity	Never

Fig. 2. The visual quark modeler

the one available in Mono (`http://www.go-mono.org`) and the one available in the .NET Framework (`http://www.microsoft.com/net`).

The compilation process is straightforward because it is basically a simple mapping between the QPL program and the equivalent IL source code that exploits the reasoning engine that the runtime platform provides.

The runtime platform is a container capable of hosting a number of quarks that can be loaded and started programmatically or from the command line.

The core of runtime platform is a reasoning engine we developed to efficiently solve constraint satisfaction and optimization problems. The discussion of the techniques we used to implement this engine is out of the scope of this paper, but it is worth mentioning that it uses a mixture of standard constraint programming algorithms, e.g., AC-2001.

The reasoning engine is multi-threaded and all quarks hosted in the same container share the same engine. This is particularly useful because the engine can handle a number of problems concurrently, with the possibility of sharing many internal structures that are possibly common to many problems. This is the reason why the CSOP solver that quarks exploit is not embedded in quarks themselves, rather the runtime platform provides it.

Figure 1 shows selected pieces of a QPL source code that can be compiled with QK compiler. The class of agents implements shop assistants for a Web-shop that can manage the configuration of PCs for customers.

QK has already been used in a number of experiments, mostly for research purposes. It has been recently adopted as the basis of a product that the company FRAMeTech S.R.L. (`http://www.frame-tech.it`) will deliver to their customer later this year. Figure 2 shows a snapshot of this product. The purpose of this product is to provide a user-friendly approach to fast developing product and service configuration systems. Such systems are meant to provide final users with a self-service mechanism for doing the configuration of complex product and services. Examples of such systems are typically used by large hardware shops to provide their customers with a Web application for configuring and then buying personalized PCs and peripherals. Another typical example regards travel agencies giving the possibility to their customers to have a fine-grained configuration of their trips. This system is basically a graphical front-end for realizing QPL programs. It allows modeling the domain of the problem using an UML class-diagram editor. Then, it allows defining all features available in a QPL program in terms of tables and expressions. Finally, it makes a GUI designer available to provide quarks with simple GUIs that final users will exploit for easily managing the problems of their quarks.

5 Conclusion

In this paper we described a set of tools that we realized to support the developer in the realization of constraint-based multiagent systems. These systems are a subset of ordinary multiagent systems because we require agents composing the system to be ascribable to CSOP solvers. This requirement was the starting point of defining a general-purpose programming language to realize agents as CSOP solvers, and a toolkit supporting the deployment of such agents in running systems. This language concentrates on modeling the problem an agent is in charge of and any other meta-level issues, e.g., governing the process of resolution, or orchestrating interactions between agents, are intentionally left implicit.

This language is concretely supported by a toolkit that provides a compiler and a runtime platform. Moreover, this toolkit supports a seamless integration of such agents with legacy systems.

Interesting future research directions regards understanding the relationship between the agent model that we propose and mentalistic agents. It is quite obvious that constraints plays a crucial part also in mentalistic agents, but a clear mapping between basic concepts of such models is still missing.

Another interesting development of this research is in the direction of understanding how inheritance of quarks can be exploited as a software engineering tool.

References

1. F. Bartak. Constraint programming – What is behind? *Procs. Int'l. Workshop on Constraint Programming in Decision and Control*, 1999.

2. C. Castelfranchi. Guarantees for autonomy in cognitive agent architecture. *Intelligent Agents*. Springer-Verlag, 1995.
3. S. Franklin and A. Graesser. Is it an agent, or just a program? *Procs. ECAI'96 Workshop on Agent Theories, Architectures, and Languages*, pages 21–36. Springer-Verlag, 1997.
4. E. C. Freuder. In pursuit of the holy grail. *Constraints*, 1(2), 1999.
5. S.-O. Hansson. *A Textbook of Belief Dynamics*. Kluwer Academic publishers, 1997.
6. M. Henz, G. Smolka, and J.Würtz. Oz – A programming language for multi-agent systems. R. Bajcsy (Ed.) *Procs.* 13th *Int'l. Joint Conference on Artificial Intelligence*, volume 1, pages 404–409. Morgan Kaufmann Publishers, 1993.
7. A. Mackworth. Quick and clean: Constraint-based vision for situated robots. *Procs. Int'l. Conference on Image Processing*, 1996.
8. M. J. Maher. Logic semantics for a class of committed-choice programs. *Procs.* 4th *Int'l. Conference on Logic Programming*, pages 858–876, 1987.
9. A. Newell. The knowledge level. *Artificial Intelligence*, 18:87–127, 1982.
10. C. L. Pape. Implementation of resource constraints in ILOG Schedule: A library for the development of constraint-based scheduling systems. *Intelligent Systems Engineering*, 3(2):55–66, 1994.
11. V. A. Saraswat and M. Rinard. Concurrent constraint programming. *Procs.* 7th *Annual ACM Symposium on Principles of Programming Languages*, 1990.
12. P. van Beek and X. Chen. CPlan: A constraint programming approach to planning. *AAAI/IAAI*, pages 585–590, 1999.
13. M. J. Wooldridge. Intelligent agents. G. Weiss (Ed.) *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, pages 27–77. The MIT Press, 1999.
14. F. Zambonelli and V. Parunak. Towards a paradigm change in computer science and software engineering: A synthesis. *The Knowledge Engineering Review*, 2004.

Debugging Agent Behavior in an Implemented Agent System

Dung N. Lam and K. Suzanne Barber

The Laboratory for Intelligent Processes and Systems,
Electrical and Computer Engineering,
The University of Texas at Austin,
Austin, TX 78712, USA
{dnlam, barber}@lips.utexas.edu
http://www.lips.utexas.edu

Abstract. As agent systems become more sophisticated, there is a growing need for agent-oriented debugging, maintenance, and testing methods and tools. This paper presents the Tracing Method and accompanying Tracer tool to help debug agents by explaining actual agent behavior in the implemented system. The Tracing Method captures dynamic runtime data by logging actual agent behavior, creates modeled interpretations in terms of agent concepts (e.g. beliefs, goals, and intentions), and analyzes those models to gain insight into both the design and the implemented agent behavior. An implementation of the Tracing Method is the Tracer tool, which is demonstrated in a target-monitoring domain. The Tracer tool can help (1) determine if agent design specifications are correctly implemented and guide debugging efforts and (2) discover and examine motivations for agent behaviors such as beliefs, communications, and intentions.

1 Introduction

There are several agent-oriented software design methodologies (e.g. GAIA [1], MaSE [2], and OMNI [3]) and development environments (e.g., JADE [4], ZEUS [5], and FIPA-OS [6]), but there are few agent-oriented methods and tools that have been created for debugging, maintaining, or testing the resulting implemented system. This paper presents the Tracing Method and accompanying Tracer tool whose purpose is to help better comprehend actual agent behavior in the implemented agent system in terms of familiar agent concepts. The objective is to ensure that an agent is performing actions for the right reasons and, if an unexpected action occurred, to help explain why an agent decided to perform the action. Due to the increasing sophistication of agent software (in particular, the autonomy, proactivity, and social features of agents) and the number of factors to consider when understanding or explaining agent behavior, comparing the implementation's actual behavior with expected behavior can be an intensive task, requiring time and knowledge about the design, implementation, and application domain. In more general terms, "concurrency, problem-domain uncertainty, and non-determinism in execution together conspire to make it difficult to comprehend the activity in a distributed intelligent system" [7]. Despite its difficulty, the task of comprehending agent behavior in the implemented agent system helps to identify undesired agent behaviors (bugs) and to gain insight on how the agents can be improved (for maintenance and testing). In addition to helping developers test and improve implementations produced by themselves or by others (e.g., open-source agent systems), the Tracing Method can help the end-user better understand the agents, and thus increase the end-user's confidence on agent decisions.

A motivation for applying agent-oriented techniques is to make the problem and solution easier to understand (i.e., by localizing beliefs and goals into autonomous agents), but after the agent system has been implemented as source code, distinguishing agent concepts (e.g., beliefs, goals, and intentions) in the implementation can become difficult as the complexity of the implementation increases. The design is specified in terms of agent concepts; however, the implementation is often specified in terms of the programming language structures, such as variables, classes, and flow-control statements. There are agent-oriented tools to generate the implementation from the design (e.g., Doi *et. al.*'s work on generating source code from AUML diagrams [8]), but there are few agent-oriented tools to extract the design concepts (i.e., agent concepts) from the implementation. This research aims to extract agent concepts from the implementation and to regain the advantages of conceptualizing the implementation in terms of agent concepts. To complement the high-level agent concepts, low-level details (e.g., belief values and communication message contents) related to the agent concepts are required for debugging the implemented agent system. Such details are made accessible by the Tracer tool, which aims to alleviate the largely manual task of comprehension during software maintenance.

This paper describes the Tracing Method to observe and interpret actual agent behaviors in terms of agent concepts, the same agent concepts that are used to describe expected agent behaviors in the design. Because agent behaviors in both the design and implementation are understood in terms of agent concepts, establishing the set of agent concepts is central to this research. Section 2 presents the proposed set of agent concepts used for describing agent behaviors. Section 3 outlines the Tracing Method for observing and extracting actual agent behaviors from the implementation's execution so that those actual behaviors can be compared with expected (or designed) agent behaviors. Section 4 demonstrates the Tracer tool in a UAV (Unmanned Aerial Vehicles) target-monitoring domain, where the implemented agents use MDPs (Markov Decision Processes) to decide their actions. The Tracer tool was found to be useful for quickly identifying and understanding the reasons for agent actions in terms of agent concepts. Section 5 describes how this research relates to existing work.

A demonstration of the Tracer tool applied to the UAV domain and to a simple multiagent system (along with example Java source code) is available for download at the website `http://www.lips.utexas.edu/~dnlam/tracer.html`.

2 Agent Concepts

Agent concepts are used in software designs to describe expected agent structure (e.g., an agent encapsulates localized beliefs, goals, and intentions) and behavior (e.g., an agent performs an action when it believes the event occurred). During and after development, agent concepts are used to abstract away from the details of the implementation and to understand and explain actual agent behavior (i.e., to answer the question "Why did agent $a1$ perform action α"). A desirable explanation could be "Action α was performed by agent $a1$ because $a1$ believed belief $b1$, which was due to the occurrence of event e in the environment, which was an expected consequence of agent $a1$ performing intention i, which was created based on belief $b2$ as communicated in a message from agent $a2$ about $a2$'s goal g." Other agent concepts that may be of interest include the roles the agents played during the interactions. An objective of this research is to automate some of the currently-manual tasks that a human must do to comprehend the implemented agent system.

This section describes agent concepts, focusing on their relationship with each other. The proposed set of agent concepts includes *goal*, *belief*, *intention*, *action*, *event*, and *message*. These agent concepts have a general definition or understanding in the agent community, but due to the variety of approaches and applications, there is no definitive representation for the agent concepts. Figure 1 illustrates the relationships among these agent concepts, and Table 1 presents the representation of agent concepts used in this research.

Table 1. Agent concept structure declarations

Agent Concept	**Constituent attributes**
event	name, preconditions, postconditions
action	agent, name, preconditions, postconditions
message	sender agent, receiver agent, subject, value
belief	agent, subject, value
goal	agent, name
intention	agent, name, goal names, belief subjects, action names

Agents are distributed, goal-oriented entities situated in an environment and encapsulate decision-making capabilities. Agents need their own *goal*(s) in order to be proactive (i.e., take initiative to achieve some goal) and autonomous (i.e., make decisions on their own based on their goals). In addition to localized beliefs about itself, agents also maintain *beliefs* about the environment, including objects situated in the environment. Beliefs are subjective representations of the state of the agent or the system and can affect many other aspects of the agent, including its goals. Using its current beliefs, an agent achieves a goal by generating an *intention* (or plan), which prescribes actions that the agent(s) intend to perform. *Actions* performed by agents and other entities can cause *events* in the environment, which agents may sense and use to update their beliefs. For

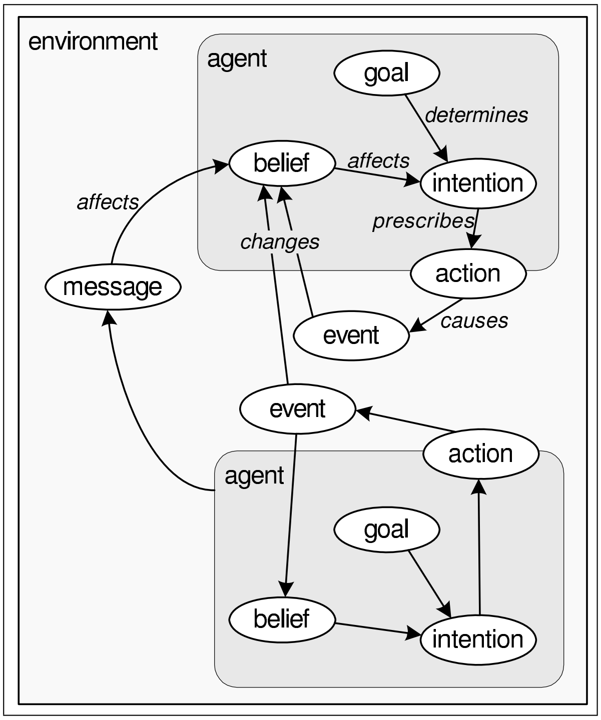


Fig. 1. Agent concepts and their relationships

explaining agent behavior, an agent's goals, beliefs, and intentions, in addition to its actions, must be considered because agents may act as expected but for undesirable reasons.

For multi-agent systems, communication is often an important factor to system performance. An agent may send *messages* to others during belief maintenance (for knowledge-sharing), during planning (for collaboration), or during schedule execution (for coordination). In terms of agent concepts, a communicated message can (directly or indirectly) affect an agent's goal, belief, intention, and/or action. Thus, an explanation of an agent action should include communicated messages that contributed to that action.

This research declares general structures for each agent concept as shown in Table 1. Unlike formal representations (e.g., goal representations for BDI agents [9]), the agent concepts are generalized so that they can be used in any implementation and to minimize the amount of effort required to apply the Tracing Method, (i.e., the effort in adding logging code). With the aim of generalizing the agent concepts, the set of attributes composing each agent concept is minimal. The attributes declared for each agent concept in Table 1 are needed to relate agent concepts with each other as illustrated in Fig. 1. For example, the attributes of a *goal* are the *agent* that wants to achieve the goal and a *name* for the goal. Other details about the goal (e.g., reward value) are not needed to relate the *goal* to the *intention* created to achieve that goal. Of course, the goal's name must be distinct from names of non-equivalent goals. Other constituent attributes of an *intention* include references to the *beliefs* that were used in the process of creating the intention and *actions* to be performed as prescribed by the intention. Note that the values for the constituent attributes are set when an agent concept is observed, or logged (this is further discussed in Section 3).

By comparing the values of the attributes constituting each agent concept, relations between agent concepts during implementation execution can be automatically formed. For example, to relate an event to an action causing that event, the action's *postconditions* must be equivalent to the event's *preconditions*. Additionally, the *name* attribute of events, actions, goals, or intentions can be compared to the *subject* attribute of messages or beliefs to denote that a message or belief can be about an event, action, goal, or intentions. Section 4 demonstrates some application-specific relations in the UAV domain. For flexibility in the Tracer tool, the structure of each agent concept can be appended for application-specific relations that are not possible with the proposed structure (i.e., there is an implicit *user object* attribute for each agent concept).

Agent concepts and their relationship with each other establish the foundation for work in automated analysis of agent system implementations. For example, in Section 4, the Tracer tool analyzes observations about an agent (in terms of agent concepts) to explain agent actions.

3 Tracing Method and Tracer Tool

Due to the inherent unpredictability of autonomous agents in an uncertain environment and the possibility of emergent behavior, Jennings stresses a need for a better understanding of how agent interaction affects an individual agent's behavior [10]. The idea of the Tracing Method is to capture uncertain, dynamic run-time data (e.g., environmental events and communicated beliefs), to observe each agent's behavioral response, and to help explain this behavior. The Tracing Method can be used repeatedly throughout the software life-cycle from the first skeleton code to the final system. Using the Tracing Method shown in Fig. 2, interpretations (models or diagrams that represent the actual agent behavior in terms of agent concepts) are created using observations resulting from the implementation's execution. These interpretations can be the same models and diagrams that result from reverse engineering (e.g., flow control, component dependence, and class inheritance models or state-chart and process-flow diagrams), or the interpretations can be relational graphs linking observations together (as is the case in this paper).

The Tracing Method and Tracer tool is being developed for agent systems that are implemented in a procedural programming language (e.g., Java, C, and C++), but they can also be used in declarative agent-oriented programming languages (e.g., AF-APL [11] and Suna *et. al.*'s mobile agent language [12]) to visualize and clarify agent behavior in the system. Currently, the Tracer tool

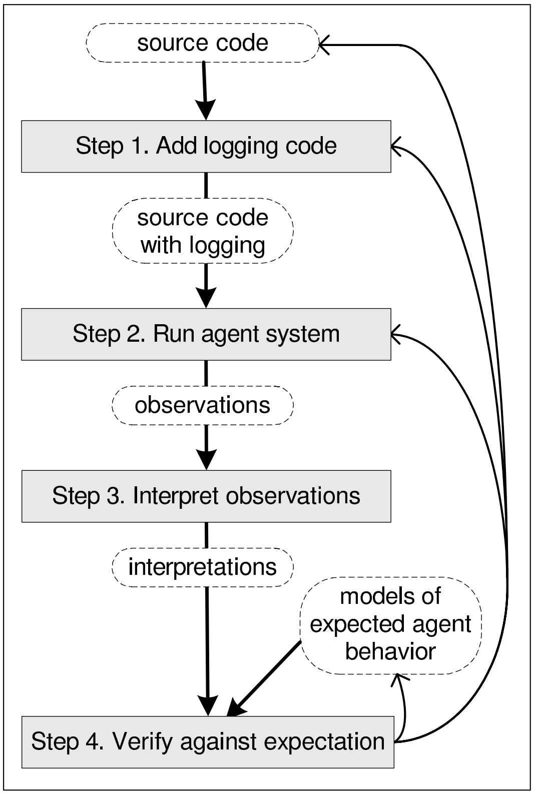


Fig. 2. Tracing Method process diagram

includes Tracing clients that allow Java and Lisp implementations to sends logs to the Tracing server.

The Tracing Method involves logging agent behavior during execution, transforming the log entries and run-time data into interpretations, and comparing those interpretations with the models of expected agent behavior. As a result, the agent concepts (e.g., beliefs about the current state of the environment) are instantiated with actual run-time data. By comparing the interpretations with the models of expected behavior, actual behavior can be verified against expected behavior. Expected behavior may be formally specified as models in design documents or informally understood and modeled by the developer(s). In either case, if there are inconsistencies between the interpretations and the expected behavior, the implementation or the expected behavior may need to be corrected. Currently, comparisons are manually performed because a formal specification for expected agent behavior has yet to be developed. However, explanations of particular observations (i.e., the observations that are inconsistent with expectations) can be generated upon user request. Details about the inconsistencies are presented to the user so that the implementation or expected behavior can be corrected. Since each observation can be traced back to a loca-

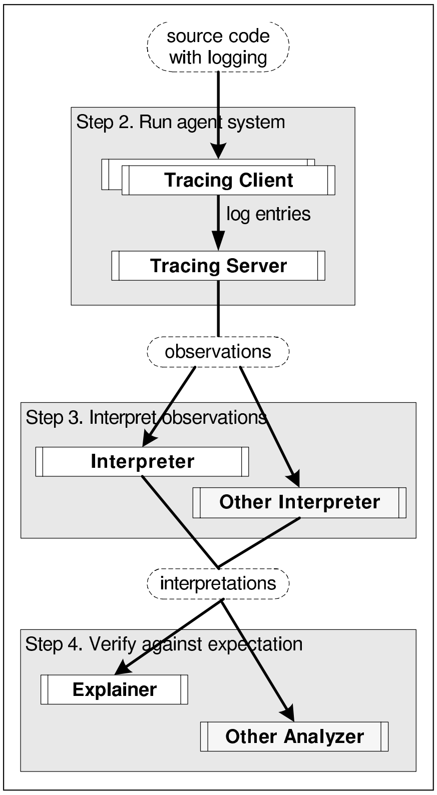


Fig. 3. Components of the Tracer tool

tion in the source code where the log entry was created (using a stack trace), correcting the inconsistency is facilitated.

Each step of the Tracing Method in Fig. 3 is described in the following subsections, accompanied by examples from the Tracer tool. To clarify which tasks in the Tracing Method have been automated and where additional features or tools can be integrated, Fig. 3 illustrates each component of the Tracer tool. The Tracer tool aims to automate the developer's task of analyzing run-time data, creating interpretations of actual agent behavior, and relating those interpretations to models of expected agent behavior. Essentially, the Tracer tool translates the procedural execution of the implementation (resulting in log entries) into declarative statements about what and when the agent believes, intends, and performs (called observations). To accommodate other analyses of the implementation, additional interpreters and analyzers can be added to the Tracer tool. The current products of the Tracer tool include behavioral and structural models (representing interpretations of the logged data based on order, duration, and other run-time attributes) and explanations of particular observations requested by the user. Since the interpretations are similar to design models, the interpretations can be compared to the original design models to ensure important aspects of the agent design have been traced. The comparison task can be performed by other automated tools that can analyze the observations or interpretations.

3.1 Step 1: Add Logging Code

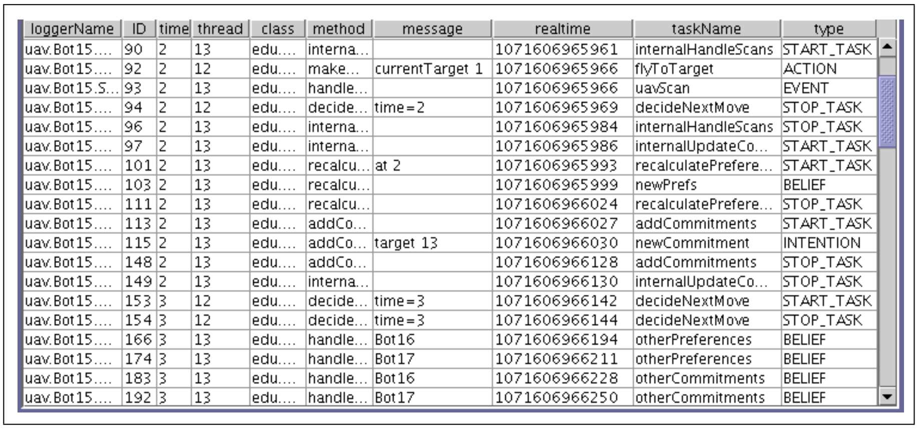
The first step of the Tracing Method is to insert code that logs run-time data about agent concepts into the source code. Unlike traditional reverse engineering tools (e.g., Gen++ [13] and DESIRE [14]), this method does not analyze code in detail, thus, it is not dependent on any specific language. Instead, run-time data about the implementation's execution is acquired by explicitly logging the data that is desired. The logging code should be added at points in the source code where an agent concept is updated or occurs, such as when an agent (1) changes a *belief* that can affect decision-making, (2) decides about its *intention* (e.g., generates a plan of action), (3) modifies its *goal*, (4) performs an *action*, and (5) sends and receives a communication *message*, as well as (6) when an *event* occurs that can affect an agents behavior. To clarify this step, the demonstration in Section 4 identifies every agent concept that is logged for the UAV domain. Currently, this step is a manual process performed by the developer, tester, or end-user, assuming the implementation is organized and structured enough so that agent concepts can be identified in the source code.

Listing 1: Example logging code in Java for an agent

```
TraceLogger logger =
   Tracer_Client.getLogger("uav.Bot"+agentID);

public void handleNewScan(DetectedTarget target){
   logger.logBelief("Target"+target.getID(), target);
   // remaining implementation . . .
}
```

Listing 1 shows an example of logging code for a belief being updated in the source code. From Table 1, the constituent attributes of a belief are the agent holding the belief, the subject the belief is about, and the value of the subject. In Listing 1, the constituent attributes of the belief are `"uav.Bot"+agentID`, `"Target"+target.getID()`, and `target`, respectively.

In addition to inserting logging code for agent concepts, logging code can be inserted at the beginning and end of code segments to denote the agent's current state or the current task (or activity) the agent is performing. Such logging code denoting the agent's tasks provides more information and context for the logged agent concepts. For example, in the first five rows shown in Fig. 4, the `flyToTarget` action and `uavScan` event occur within the `internalHandleScans` task because they appear between its `START_TASK` and `STOP_TASK`. For more details about logging agent tasks, an earlier paper [15] describes how such con-



loggerName	ID	time	thread	class	method	message	realtime	taskName	type
uav.Bot15....	90	2	13	edu....	interna...		1071606965961	internalHandleScans	START_TASK
uav.Bot15....	92	2	12	edu....	make...	currentTarget 1	1071606965966	flyToTarget	ACTION
uav.Bot15.S...	93	2	13	edu....	handle...		1071606965966	uavScan	EVENT
uav.Bot15....	94	2	12	edu....	decide...	time=2	1071606965969	decideNextMove	STOP_TASK
uav.Bot15....	96	2	13	edu....	interna...		1071606965984	internalHandleScans	STOP_TASK
uav.Bot15....	97	2	13	edu....	interna...		1071606965986	internalUpdateCo...	START_TASK
uav.Bot15....	101	2	13	edu....	recalcu...	at 2	1071606965993	recalculatePrefere...	START_TASK
uav.Bot15....	103	2	13	edu....	recalcu...		1071606965999	newPrefs	BELIEF
uav.Bot15....	111	2	13	edu....	recalcu...		1071606966024	recalculatePrefere...	STOP_TASK
uav.Bot15....	113	2	13	edu....	addCo...		1071606966027	addCommitments	START_TASK
uav.Bot15....	115	2	13	edu....	addCo...	target 13	1071606966030	newCommitment	INTENTION
uav.Bot15....	148	2	13	edu....	addCo...		1071606966128	addCommitments	STOP_TASK
uav.Bot15....	149	2	13	edu....	interna...		1071606966130	internalUpdateCo...	STOP_TASK
uav.Bot15....	153	3	12	edu....	decide...	time=3	1071606966142	decideNextMove	START_TASK
uav.Bot15....	154	3	12	edu....	decide...	time=3	1071606966144	decideNextMove	STOP_TASK
uav.Bot15....	166	3	13	edu....	handle...	Bot16	1071606966194	otherPreferences	BELIEF
uav.Bot15....	174	3	13	edu....	handle...	Bot17	1071606966211	otherPreferences	BELIEF
uav.Bot15....	183	3	13	edu....	handle...	Bot16	1071606966228	otherCommitments	BELIEF
uav.Bot15....	192	3	13	edu....	handle...	Bot17	1071606966250	otherCommitments	BELIEF

Fig. 4. Log entries for agent `Bot15` in the UAV domain

textual logging code can result in state-chart and process-flow diagrams (other representations of software behavior), which are then verified.

Since only agent concepts are logged, this logging approach requires only high-level functional and structural knowledge of the implementation. If there is insufficient or erroneously-placed logging code, these errors will manifest themselves as specific inconsistencies between the interpretations and the models of expected behavior in the fourth step in Fig. 2. The developer can use the identified inconsistency to determine if logging code should be added or modified.

3.2 Step 2: Run Agent System

The second step of the Tracing Method is to execute the agent system so that the Tracer tool can collect run-time data, such as when and where the logging code was executed. A logging mechanism based on the Java Logging Framework [16] has been implemented. When the logging code executes, log entries are created from run-time data (e.g., what the agents believe and intend, what actions are being performed, and what events are occurring in their environment) and are sent by the Tracing Client to the Tracing Server locally or across a network (see Fig. 3).

Log entries for agent concepts and task activities are transformed into generic log entries so that they can all be handled by the same tools. For example, Fig. 4 displays all types of log entries on a single display. Figure 4 shows log entries as rows and the corresponding run-time data (e.g., timestamp and process id) as columns for an agent in the UAV domain. Each column is described as follows:

Loggername : context of the log entry identifying an agent or the simulator (e.g., `uav.Bot15` or `uav.Sim`) or a subcomponent within an agent (e.g., `uav.Bot15.planner`);
ID : unique identifier for the log entry;
Time : simulation time at which log entry was created;

Thread : execution thread id of the process in which the log entry occurred;
Class, Method : class and method in which the logging code executed (a full stack trace with source code line numbers is also available but not shown in the figure);
Message : additional free-form details about the log entry for human readability;
Realtime : real time at which log entry was created;
Taskname : name for observation or task;
Type : type of observation (e.g., action, event, belief) or task.

Most of the data (e.g., time, thread, class, method, and realtime) are acquired at run-time and appended to the log entry.

Due to the large amount of data, the log files need to be pre-processed and organized before they are analyzed. Log file utilities within the Tracing Server were created to sort, splice, and merge the log files so that log entries are organized correctly and interpretations accurately represent the implementation. To organize the log files, the loggerName can refer to an entire agent or a component within the agent. For each unique loggerName, there is a single log file. Thus, a thread that operates across several agent components may write to several log files. Since there may be several execution threads logging to a single file, the log file needs to be spliced into separate log files for each execution thread. For instance, in Fig. 4, the log entry with ID 94 was logged by thread 12, while most of the other entries were logged by thread 13.

3.3 Step 3: Interpret Observations

The third step is to construct interpretations that can be compared with models of expected behavior. There are several ways to automatically interpret the observations listed in Fig. 4, depending on what type of information is desired and what is being analyzed. For example, using the timestamp of observations, a state-transition diagram can be generated by one of the Tracer tool's interpreters (see [15] for an example). Additional interpreters can be added to produce other types of interpretations, including a time-plot of agent activities, data flow graphs, and message sequence charts. Each type of interpretation can be used to verify certain aspects of the agent system implementation as described in the next step. Given the desired interpretation type (in this case, it is a relational graph), the Tracer tool can generate the interpretation by processing the observations during run-time or after the execution has completed. Being able to monitor the agents during run-time offers an additional visualization of the running agent system.

To generate relational graphs used in the UAV demonstration, rules to relate agent concepts to each other are applied to the observations. The purpose for the rules is to form the relations illustrated in Fig. 1. For example, one rule states that if a message m from agent a contains the same information as belief b, then that message m is causally linked to that belief b. Another rule states that if intention p has belief b's subject in its list of belief subjects, then belief b affects

p. The resulting directed graph of these two rules implies that intention p was affected by belief b, which was a result of agent a sending message m.

To reduce the effort of applying the Tracing Method, the idea behind the interpreter is to automatically generate informative representations of agent behaviors from simple, reusable rules. Essentially, the rules compare the constituent attributes of agent concepts (e.g., name, subject, preconditions, and postconditions) in order to associate one agent concept with another. Because the agent concept structures were designed to be general, the rules can be reused in other agents with similar behaviors. Application-specific rules can be created by hand or generated by a pattern discovery mechanism. Future work will consider automatically generating the rules based on patterns in the list of observations.

3.4 Step 4: Verify Interpretations

The fourth step is to verify the interpretations against the models of expected agent behavior. There are several ways to verify the interpretations depending on the interpretation type. For example, state-transition diagrams and message sequence charts can be directly compared to expected behavior expressed as state-chart diagrams and communication protocol diagrams in design documents. Currently, due to the lack of a formal specification of agent behavior, verifying the interpretations is a manual step – a human must determine whether the interpretations are consistent with expectations. However, the Tracer tool's Explainer can assist the user in identifying the causes of unexpected behavior. Given the relational graph from Step 3, an explanation describing the observations relating to an agent action (or any observed agent concept) can be examined to ensure that an agent is performing the action for the right reasons. Section 4 demonstrates how an explanation is created.

To allow for other analyses of different interpretation types, additional analysis tools can be plugged into the Tracer tool (see Fig. 3). Possible analyses include checking safety and liveness properties about the execution trace, verifying that the agents are following communication protocols, and locating computational bottlenecks.

If the interpretations are inconsistent or seem erroneous with respect to expectations, (1) logging code may need to be added or corrected due to missing or misplaced observations, (2) the agent system may need to be executed multiple times to verify interpretation variations due to nondeterminism, (3) an implementation bug may need to be corrected, and/or (4) expected agent behavior may need to be updated. This is why there are arrows pointing from interpretations to previous steps or objects in Fig. 2. Since each high-level observation contains a low-level stack trace denoting where and in what context the logging code was executed in the source code, correcting inconsistencies is facilitated. The result of the Tracing Method is a set of verified interpretations of the implemented agents' behaviors in terms of agent concepts. As a side effect, the source code is sparsely documented with logging code that identifies important points in the code for understanding the implemented agents' behaviors.

4 Tracing the UAV Domain

To demonstrate the Tracing Method, the Tracer tool will be used to trace agents, each controlling a UAV (Unmanned Aerial Vehicle), in an implemented agent system for the UAV target-monitoring domain. The agents' overall objective is to ensure that all mobile targets are being scanned (by flying to the target's believed location and finding the target) with minimal time from when the target was last scanned. Each agent shares with other agents its own preferences about which targets it prefers to monitor. Each agent individually decides which targets it intends to scan (referred to as its committed targets) so that all targets are being scanned as frequently as possible using a Markov Decision Process (MDP) with a value function that considers distances from targets, targets' last scan times, and other agents' target preferences and commitments. Each agent's MDP model is updated as the agent receives new information about targets and other agents, thus affecting the agent's decision about which targets to monitor. Since there are a lot of factors for each agent to consider and the decision-making process occurs frequently, checking agent behavior would be facilitated by using the Tracing Method and Tracer tool.

The first step is to add logging code to the source code for each agent concept. To demonstrate that a low-level understanding of the implementation was not necessary, the person adding the logging code was not intimately familiar with the source code for the simulation or the agents and asked the developer only high-level questions concerning the abstractions from code to agent concepts. Note that the simulation was provided by Metron, Inc., as a contribution to the DARPA TASK project, the agents were programmed by a developer in our lab, and the logging code was added to the simulation and the agents by the author of this paper. The following lists specific instances for each agent concept in the UAV domain:

Message : messages about preferences, commitments, and scanned targets;
Belief : beliefs about an agent's own preferences, commitments, and scanned targets and, via communicated messages, beliefs about other agents' preferences, commitments, and scanned targets;
Desire : (static) minimize time between scans for all targets;
Intention : ordered list of committed targets;
Action : fly to target, spiral (to search for target), and stop;
Event : a target is scanned (if the agent is within range as determined by the simulation).

Once logging code was inserted for each of these agent concepts, the second step is to execute the implementation. The simulation was executed with fifteen targets (`0` to `14`) and three agents (`Bot15`, `Bot16`, and `Bot17`). During execution, the Tracing Clients send log entries (see Fig. 4) to the Tracing Server, which translates the log entries into observations. Table 2 partially lists the observations for agent `Bot15` in human readable format.

For each observation, the table shows the type of observation, the simulation time the log occurred, a unique identifier for the observation, and run-time data

Table 2. Partial list of observations for agent `Bot15`

Type	Time	ID	Run-time data
Belief	0	18	initTargets (3 7 2 0 14 6 1 10 5 13 9 11 4 8 12)
Belief	0	19	initTargetLocations ((151.26 203.46) (536.57 517.74) ... (55.01 77.03))
Action	1	31	stop
Event	1	33	uavScan ( )
Belief	1	42	myPreferences (1 13 6 9 5 10 2 3 12 8 4 11 0 14 7)
Intention	2	67	addCommitment (target 1), intention=(1)
Action	2	92	flyToTarget (1)
Event	2	93	uavScan ( )
Belief	2	103	myPreferences (1 13 6 9 5 10 2 3 12 8 4 11 0 14 7)
Intention	2	115	addCommitment (target 13), intention=(1 13)
Message	3	160	messageReceived from Bot16 sentAtTime 2 (about Bot16 preferences (10 1 6 5 8))
Belief	3	166	otherPreferences Bot16 (10 1 6 5 8)
Message	3	172	messageReceived from Bot17 sentAtTime 2 (about Bot17 preferences (1 6 5 10 9))
Belief	3	174	otherPreferences Bot17 (1 6 5 10 9)
Message	3	180	messageReceived from Bot16 sentAtTime 2 (about Bot16 commitments (10))
Belief	3	183	otherCommitments Bot16 (10)
Message	3	189	messageReceived from Bot17 sentAtTime 2 (about Bot17 commitments (1))
Belief	3	192	otherCommitments Bot17 (1)
Event	4	207	uavScan ( )
Event	4	228	uavScan ( )
Message	4	241	messageReceived from Bot17 sentAtTime 4 (about Bot17 preferences (9 3 6 2 5))
Belief	4	244	otherPreferences Bot17 (9 3 6 2 5)
Message	5	291	messageReceived from Bot16 sentAtTime 4 (about Bot16 preferences (8 10 11 3 2))
Belief	5	293	otherPreferences Bot16 (8 10 11 3 2)
Message	5	307	messageReceived from Bot17 sentAtTime 4 (about Bot17 commitments (1 9))
Belief	5	309	otherCommitments Bot17 (1 9)
Message	5	316	messageReceived from Bot16 sentAtTime 4 (about Bot16 commitments (10 8))
Belief	5	318	otherCommitments Bot16 (10 8)
Event	5	323	uavScan ( )
Event	7	353	uavScan ( )
Event	7	361	uavScan ( )
Event	8	393	uavScan ( )
Event	9	415	uavScan ( )
Event	10	440	uavScan ( )
Event	11	460	uavScan ( )
Event	12	484	uavScan (1)
Belief	13	510	scannedTarget (1)
Action	14	520	flyToTarget (13)
Belief	15	566	myPreferences (13 10 5 12 4 0 1 7 14 3 2)
Intention	15	594	addCommitment (target 10), intention=(13 10)
...	...	...	...

pertaining to the observation. The run-time data offers details such as what was believed, what action occurred, or what commitments were made by agent `Bot15`. Note that the observations are chronologically ordered by the simulation time and that the ID coincides with this temporal ordering. In this agent implementation, only messages received by `Bot15` are listed for conciseness, since messages sent by `Bot15` do not directly affect its own decisions. Also, no goal type observations are listed because all agents have a static goal of minimizing the time between scans for each target.

A glance through the table shows an unexpected (or at least undesired) behavior. The `uavScan` event, which signifies that an agent scans its current location for targets, was found to occur more than once per simulation timestep (e.g., observation 207 and 228 in timestep 4). Since the simulation only provides new scans once per timestep, the duplicate event is unnecessary. After consulting with the developer, this repetition occurs because the simulation's execution thread is running faster than the agent's execution thread. As a result, to make up for missed scans in previous timesteps, the agent performs multiple scans per timestep. For example, in Table 2, the agent does not scan in timestep 6, so the scan event is repeated in timestep 7. This undesired behavior does not adversely affect the overall performance of the agents with respect to its goal. However, this identified inefficiency may affect real-time performance as the number of agents increase or as the agents become slower than the simulation.

The third step is to interpret the observations by generating relational graphs. Rules, including the general rules mentioned in Section 3, were applied to the observations in order to create directed graphs. The general and application-specific rules that were used are listed below. The algorithm applies the appropriate rules to each new observation that appears and searches backward (temporally) to find the latest previous observation that satisfies the rule antecedent.

General rules:

– If belief b occurs after message m and their subjects and values are equivalent, then m affected b.
– If message m occurs after belief b and their subjects and values are equivalent, then b's occurrence caused m to be sent.
– If event e occurs after action a and e's precondition is equivalent to a's postcondition, then a caused e.
– If belief b occurs after event e and b's subject is equivalent to e's name and b's value is equivalent to e's postcondition, then e caused b.
– If an action a's name is contained in a previously observed intention i's action names, then i prescribed a.
– If intention i occurs after belief b and i's belief subjects includes b's subject, then belief b influenced the intention i.
– If message m occurs after an action, intention, or goal o whose name is equivalent to m's subject, then o influenced m.

Application-specific rules:

– If intention $i2$ occurs after intention $i1$, then $i1$ influenced $i2$.
– If a belief $b2$ occurs after the last `myPreferences` belief $b1$ and before a `myPreferences` belief $b3$, then $b2$ affects `myPreferences` belief $b3$.
– If a `flyToTarget` action a occurs after a `scannedTarget` belief b for different targets, then a is cause by b (i.e., the agent believes it has scanned its previous target and is pursuing its next target as prescribed by its intention).

More complicated application-specific rules can be created to relate more than two observations together, but for the general rules, straightforward rules are preferred for better reuse. The rules represent the background knowledge used

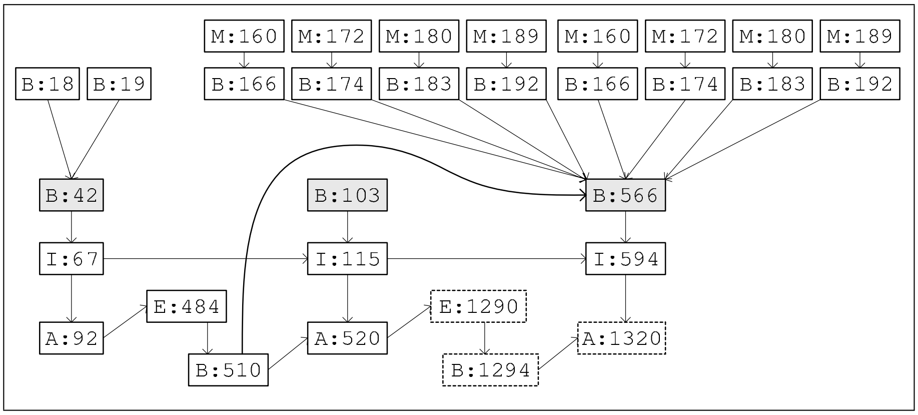


Fig. 5. Relational graph for agent `Bot15`

by the users to generate explanations. Such background knowledge needs to be represented in the Tracer tool in order to automate explanation generation. Currently, these rules are manually defined and given as input to the Tracer tool. For future work, the tool will discover patterns in the observations and suggest to the user rules that relate agent concepts to each other; thus, further automating the comprehension tasks.

Figure 5 illustrates the relational graph generated from the observations in Table 2. Each node in the graph is labeled with the first letter of the observation type (i.e., `B`=belief, `I`=intention, `A`=action, `E`=event, and `M`=message) and the unique id of the observation, so it can be referenced in Table 2. Each edge represents a source node causing (or influencing) the destination node. The *belief* nodes B:18 and B:19 show that agent `Bot15` processes initial data about the targets and their locations. Based on only those beliefs, `Bot15` creates its initial target preferences labeled B:42.

The shaded belief nodes represent `myPreferences` that have been calculated using `Bot15`'s current beliefs about the targets and other agents' target preferences and commitments. For the target preferences in B:42 and B:103, only the initial beliefs were used since the other agents have not yet communicated their preferences or commitments. However, in B:566, `Bot15` takes advantage of several beliefs (about other agents' preferences and commitments) that were created from communicated messages (i.e., M:160, M:172, M:180, etc.) as shown in Fig. 5 and detailed in Table 2.

The fourth step is to analyze actual agent behavior to insure agents are behaving as expected. In doing so, an end-user can gain a better understanding of what the agent is doing and why. A description of what the agent is doing is described below, followed by a description of how explanations are generated by the Tracer tool's Explainer.

Based on preferences in B:42, the agent makes a commitment represented by the *intention* node I:67 (i.e., `Bot15` adds commitment to scan `target 1`). Next, based on *intention* I:67, the agent performs an *action* A:92 (i.e., `Bot15`

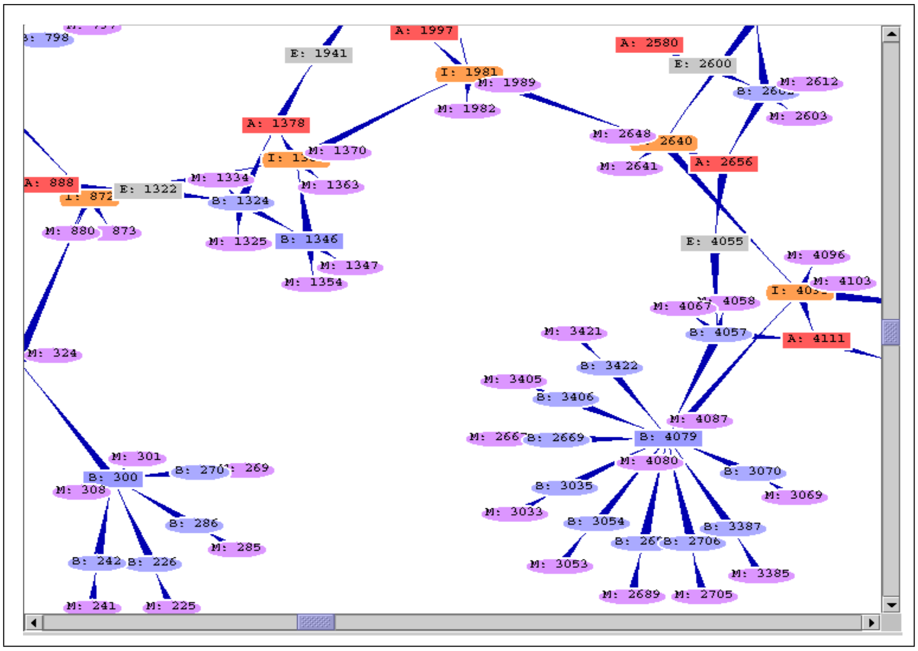
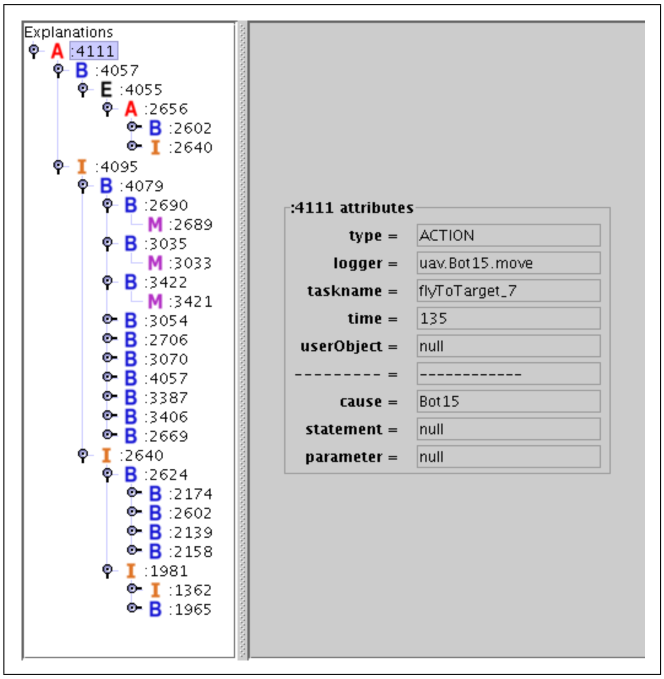


Fig. 6. Relational graph from Tracer tool

flies to `target 1`'s believed location). This series of observations can be easily followed in Table 2. Before *event* E:484 (i.e., `Bot15` scans `target 1`) occurs at timestep 12, the agent recalculates its preferences to create B:103 (i.e., `Bot15`'s new preferences are `( 1 13 6 9 ... )`) at timestep 2, which is not different from B:42 (as seen in Table 2) because there were no new beliefs to consider between B:42 and B:103. The reason the agent recalculates its preferences is to add its next target, which is target 13 as shown by *intention* I:115 in the table. Reasonably, the agent does not perform *action* A:520 (i.e., `Bot15` flies to target 13's believed location) until it believes B:510 (i.e., `Bot15` has scanned `target 1`) as shown in the graph.

A relational graph can present important information that may not be as obvious when presented as a list or table. The Tracer tool's Explainer can assist the user in analyzing the relational graph by generating explanations of specified observations. Given some observation to explain (e.g., an agent action), the explanation is generated by following the incoming edges of the node that represents the given observation. For example, the graph in Fig. 5 clearly shows that *action* A:520 is prescribed by the *intention* I:115, created before communication with other agents. In the table, however, since A:520 chronologically occurs after the communications with other agents, the action A:520 can be misconstrued to be influenced by those communications. Such information can save time and effort in trying to debug the implemented system. Figure 6 and Figure 7 show snapshots of the relational graph and an example explanation from the Tracer tool.



```
Explanations
A :4111
  B :4057
    E :4055
      A :2656
        B :2602
        I :2640
  I :4095
    B :4079
      B :2690
        M :2689
      B :3035
        M :3033
      B :3422
        M :3421
      B :3054
      B :2706
      B :3070
      B :4057
      B :3387
      B :3406
      B :2669
    I :2640
      B :2624
        B :2174
        B :2602
        B :2139
        B :2158
      I :1981
        I :1362
        B :1965
```

:4111 attributes

type =	ACTION
logger =	uav.Bot15.move
taskname =	flyToTarget_7
time =	135
userObject =	null
--------- =	------------
cause =	Bot15
statement =	null
parameter =	null

Fig. 7. Explanation of action A:4111 in Fig. 6

As demonstrated, the graph provides a quick way to understand (and ask questions about) the operations of the agent without having to understand the implementation in-depth. The graph can also help answer questions, such as "Why did `Bot15` intend I:594", by narrowing down the set of beliefs that influenced the agent to decide on I:594. Additionally, some patterns of behavior are more easily discovered by the human user in graph form. For example, in Fig. 5, the nodes with dotted outline represent subsequent observations that relate *action* A:520 and *intention* I:594 (namely, the post-conditions of A:520 must be true before the next *action* A:1320 in I:594 can be performed), completing the pattern of behavior. More high-level behavioral patterns can be seen in Fig. 8, where there are clusters of belief and message nodes surrounding intention nodes and the clusters are connected by action nodes. Not surprisingly, such a pattern resembles the classical sense-reason-act cycle used in artificial intelligence.

Given patterns of behavior for an agent, anomalous behavior can be quickly identified as subgraphs that are not similar to the pattern. For example, Fig. 9 shows anomalies (visualized as dangling nodes on the right side of the figure) in behavior for an earlier version of the agent. Such anomalies can identify possible bugs in the system or unexpected changes in agent behavior. Given this information, the user can investigate the cause of the bug at that anomalous observation or add a new rule (to the background knowledge) to account for the new behavior. A future feature in the Tracer tool will be automated graph pattern discovery and anomalous behavior detection.

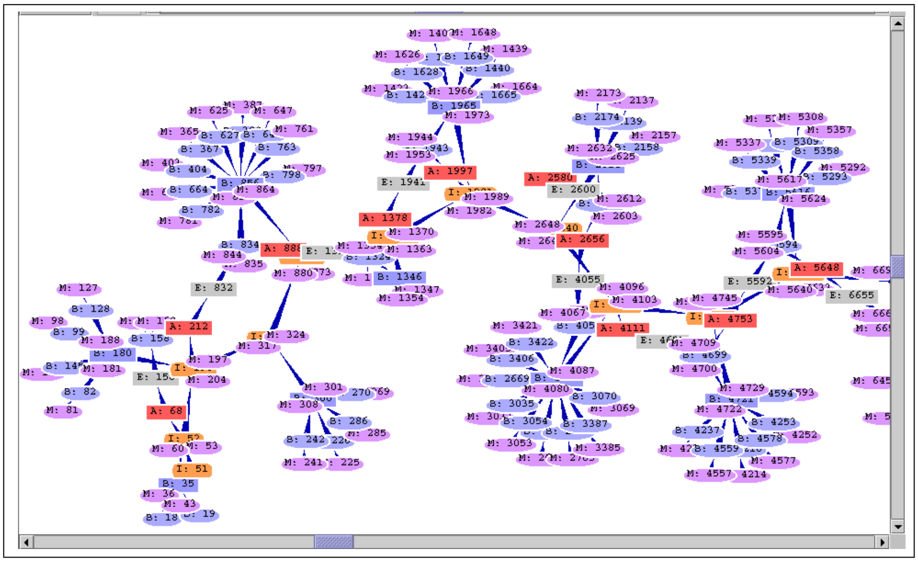


Fig. 8. Patterns in relational graph

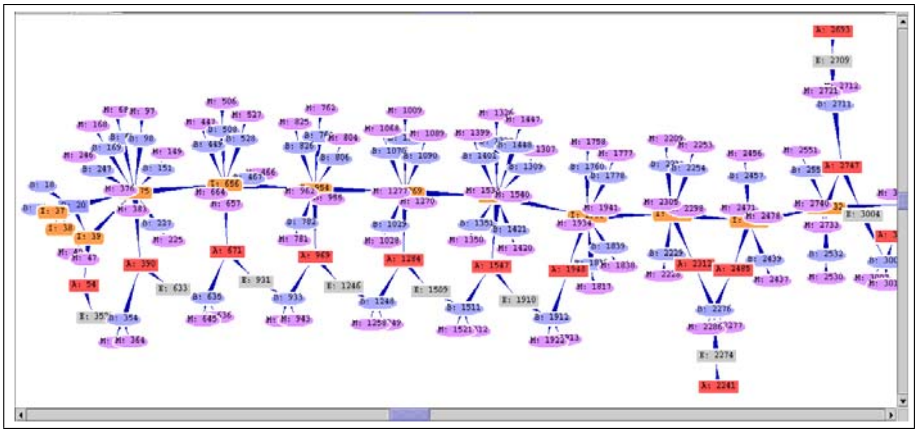


Fig. 9. Anomalies (on right-side) in relational graph for a different agent

5 Related Work

This section discusses the limitations of two popular approaches for verifying software behavior (model-checking and reverse engineering) when applied to agent-based implementations. This research addresses the limitations in using these verification approaches.

Model-checking performs a thorough search through a high-level model of an implementation to find undesired behaviors and to ensure desired behaviors as specified by the user. To verify agent behaviors using model-checking, (1) observed agent behavior (as understood by the user) and properties to be verified are translated into a format suitable for a model-checking tool, (2) the model is checked by the tool (provided the state space size is manageable), and (3) results from model-checking are interpreted and related back to the actual system (i.e., mapping a property violation back to the implementation).

For software developers who are not experts in model-checking, the translation and interpretation steps may be particularly challenging, and even for a seemingly trivial system, the large state space may be unmanageable. To reduce the learning curve associated with model-checking, software engineering researchers have focused on tools and methods to enable model-checking of high-level models (e.g., Petri-nets [17], UML diagrams [18], and architecture representations [19]). While these approaches have helped reduce the translation and interpretation barriers, they do not leverage software models that incorporate agent-related abstractions (i.e., agent concepts) and do not facilitate translating actual agent behavior from the implementation to models to be checked.

Bordini *et. al.* applied model-checking to reactive-planning agents implemented in the BDI logic programming language AgentSpeak by translating the implementation into a finite-state model that can be verified using the Spin model-checker [20]. Though promising for agent systems implemented in logic or for applications requiring formal verification, the use of model-checking in the numerous procedural (infinite-state) implementations requires effort in abstracting and translating the source code into a checkable model and may not be practical.

Edmunds points out the insufficiency of formal methods and the need for an experimental approach for understanding multi-agent systems [21]. This research offers the Tracing Method as an experimental approach to analyzing agent behavior, unlike model-checking. First, to minimize translation errors due to misunderstanding the implementation, agent behavior is constructed from the log of actual agent beliefs, intentions, and actions. Additionally, the user is only required to know where agent concepts are updated in the source code so that logging code can be added. Second, to avoid the large state space representing the aggregate behavior of an agent-based system, the Tracing Method analyzes agent behavior scoped by the set of scenarios through which the implementation is executed. The scope of scenarios for tracing can be iteratively modified as the typical development effort is iterative. Third, relating the analysis results back to the implementation is facilitated because each observation of agent behavior can be traced to an exact location in the implementation. There are a number of possible points of failure, but future work hopes to provide guidelines or (fully or partially) automate some of the steps in the Tracing Method.

Traditional software reverse engineering tools (e.g., Gen++ [13] and DESIRE [14]) analyze the implementation at the source code level and produce models of the implementation (e.g., flow control, component dependence, and class inheritance models) that are detailed representations of *what* is happening in the implementation. With the increase in complexity of agent systems, it becomes very difficult to get a comprehensive system view of the implementation using traditional tools due to the number of software components and low-level interactions involved. Without having to analyze the source code in detail, the Tracer tool automates the analysis of *what* is happening and helps to explain *why* such behavior is happening using high-level agent concepts (e.g., beliefs, goals, and communication messages). Since an agent system is conceptualized and designed using agent concepts, comprehension and verification of agent behaviors in the implemented system for debugging, testing, and maintenance should also use agent concepts.

6 Summary

As agent systems become more complex and sophisticated, there is a growing need for agent-oriented methods and tools to debug, maintain, and test agent software. This paper presents the Tracing Method and accompanying Tracer tool to help (1) verify actual agent behavior in the implemented system against expected (or designed) agent behavior and (2) understand the implemented agent system in terms of the same agent concepts used in the software design. Agent concepts are used to describe agent structure (e.g., an agent encapsulates localized *beliefs*, *goals*, and *intentions*) and behavior (e.g., an agent performs an *action* when it believes the *event* occurred). The Tracing Method captures dynamic run-time data during implementation execution, interprets the data as observations of actual agent behavior, and analyzes those interpretations.

The Tracer tool facilitates the ability (1) to determine if agent design specifications are correctly implemented and guide debugging efforts and (2) to examine and discover motivations, such as beliefs, intentions, and communicated messages, for agent behaviors. As demonstrated in the target-monitoring UAV domain, the Tracer tool assists in gaining insight into agent behavior by automating the process of generating interpretations of the implementation execution and presenting observed agent behavior in terms of agent concepts. In addition, the Tracing Method establishes general structures for agent concepts that can be used in most agent system implementations, thus, moving away from ad hoc debugging techniques.

This research proposes a method and tool to create models of agent behavior that not only describe *what* is occurring in the implementation but also *why* a respective agent behavior occurred (e.g., agent X took action a because of belief b). To enable such explanations, the Tracing Method requires only a high-level understanding of where agent concepts are modified in the implementation. In this regard, the behavior of agents in unfamiliar agent systems can be quickly understood. Overall, the Tracing Method and Tracer tool sheds light on how agents actually behave and how agent behavior in the implementation can be improved.

Acknowledgements

This research was funded in part by the Defense Advanced Research Projects Agency and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0588. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed on implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

References

1. Wooldridge, M., Jennings, N.R., Kinny, D.: The GAIA methodology for agent-oriented analysis and design. Journal of Autonomous Agents and Multi-Agent Systems **3** (2000) 285–312
2. DeLoach, S.A., Wood, M.F., Sparkman, C.H.: Multiagent systems engineering. International Journal of Software Engineering and Knowledge Engineering **11** (2001) 231–258 World Scientific Publishing Company.
3. Dignum, V., Vazquez-Salceda, J., Dignum, F.: OMNI: Introducing social structure, norms and ontologies into agent organizations. In: Second International Workshop on Programming Multi-Agent Systems at the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems, New York, NY (2004) 91–102
4. JADE: Java agent development framework. http://sharon.cselt.it/projects/jade/ (2000)
5. Nwana, H.S., Ndumu, D.T., Lee, L.C., Collis, J.C.: ZEUS: A toolkit for building distributed multi-agent systems. In Etzioni, O., Muller, J.P., Bradshaw, J.M., eds.: Third International Conference on Autonomous Agents, Seattle, WA, ACM Press (1999) 360–361
6. Poslad, S., Buckle, P., Hadingham, R.: The FIPA-OS agent platform: Open source for open standards. In: Fifth International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents, Manchestor, UK (2000) 355–368
7. Gasser, L., Braganza, C., Herman, N.: MACE: A flexible testbed for distributed AI research. In Huhns, M.N., ed.: Distributed Artificial Intelligence. Morgan Kaufmann, San Mateo, CA (1987) 119–152
8. Doi, T., Yoshioka, N., Tahara, Y., Honiden, S.: Bridging the gap between AUML and implementation using FOPL. In: Second International Workshop on Programming Multi-Agent Systems at the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems, New York, NY (2004) 69–78
9. Braubach, L., Pokahr, A., Lamersdorf, W., Moldt, D.: Goal representation for BDI agent systems. In: Second International Workshop on Programming Multi-Agent Systems at the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems, New York, NY (2004) 9–20
10. Jennings, N.R.: Agent-oriented software engineering. In: Lecture Notes in Computer Science: Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World : Multi-Agent System Engineering (MAAMAW-99). Volume 1647. (1999) 1–7
11. Ross, R., Collier, R., O'Hare, G.M.: AF-APL – bridging principles & practice in agent oriented languages. In: Second International Workshop on Programming Multi-Agent Systems at the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems, New York, NY (2004) 21–33

12. Suna, A., Fallah-Seghrouchni, A.E.: A mobile agents platform: Architecture, mobility and security elements. In: Second International Workshop on Programming Multi-Agent Systems at the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems, New York, NY (2004) 57–66
13. Devanbu, P.T.: GENOA- a customizable, language- and front-end independent code analyzer. In: Fourteenth International Conference on Software Engineering, Melbourne, Australia (1992) 307–319
14. Biggerstaff, T.J., Mitbander, B.G., Webster, D.: Program understanding and the concept assignment problem. Communications of the ACM **37** (1994) 72–83
15. Barber, K.S., Lam, D.: Enabling abductive reasoning for agent software comprehension. In: 18th International Joint Conference on Artificial Intelligence Workshop on Agents and Automated Reasoning, Acapulco, Mexico (2003) 7–13
16. Sun Microsystems, Inc.: Java Logging API. http://java.sun.com/j2se/1.4/docs/guide/util/logging (2002)
17. Grahlmann, B., Pohl, C.: Profiting from SPIN in PEP. In: SPIN '98 Workshop. (1998)
18. Bose, P.: Automated translation of uml models of architectures for verification and simulation using SPIN. In: IEEE International Conference on Automated Software Engineering. (1999) 102–109
19. Barber, K.S., Graser, T.J., Holt, J.: Providing early feedback in the development cycle through automated application of model checking to software architectures. In: 16th International Conference on Automated Software Engineering, San Diego, CA (2001) 341–345
20. Bordini, R.H., Fisher, M., Pardavila, C., Wooldridge, M.: Model checking AgentSpeak. In Rosenschein, J.S., Sandholm, T., Michael, W., Yokoo, M., eds.: Second International Joint Conference on Autonomous Agents and Multi-Agent Systems, Melbourne, Australia, ACM Press: New York (2003) 409–416
21. Edmonds, B., Bryson, J.: The insufficiency of formal design methods. Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (2004) 938–946

A Mobile Agents Platform: Architecture, Mobility and Security Elements

Alexandru Suna and Amal El Fallah-Seghrouchni

LIP6 - CNRS UMR 7606, University of Paris 6
{Alexandru.Suna, Amal.Elfallah}@lip6.fr

Abstract. This paper presents a mobile agents platform called SyMPA, compliant with the specifications of the MASIF standard from the OMG, that supports both stationary and mobile agents implemented using the high-level agent-oriented programming language CLAIM. Agents designed thanks to CLAIM are endowed with cognitive capabilities, are able to communicate with other agents and are mobile. The primitives of mobility are inspired from the ambient calculus. The paper is focused on SyMPA's architecture, mobility, implementation and security elements.

Keywords: Agents, Mobility, Agent platform, Security.

1 Objectives

The emergence of autonomous agents and multi-agent systems (MAS) technology is one of the most exciting and important event occurred in the computer science and the artificial intelligence during the 1990s. In the last years, the mobile agents paradigm became popular as a natural and flexible way to manage latency and bandwidth in the distributed systems over the network [15].

Until now, the focus of MAS community has been on the development of informal and formal tools (e.g. consortium such as FIPA and OMG have tended to propose a wide range of standards to cover the main aspects of MAS engineering), concepts (e.g. concerning mental or social attitudes, communication, cooperation, organization, planning, mobility) and techniques (e.g. AUML, modal languages such as BDI[20]) in order to be able to analyze and specify MAS. Unfortunately, less attention has been paid to the development of programming languages and tools which can effectively support MAS implementation. Indeed, for a larger use of MAS paradigm in real-world applications, specific programming languages, agent oriented, are needed.

Our long time work is motivated by two main objectives:

1. Usually, when people are developing MAS applications, they are using agent concepts for the design phase but they are using objects for the implementation. Our first objective was to propose a declarative language (CLAIM) that helps the MAS developer to reduce the gap between the design and the implementation phases and frees the designer from the most implementation aspects, *i.e.* the

designer should think and implement in the same paradigm (namely through agents). Also, this language meets the requirements of mobile computation which becomes popular due to the recent developments in the mobile code paradigm. The language must have a well defined operational semantics in order to make the verification of MAS possible.

2. The second objective is to offer a platform that supports MAS written in CLAIM, called SyMPA, compliant with the specifications of the MASIF [18] standard from the OMG. Such a platform must offer all the mechanisms needed for the design and the secure execution of a distributed MAS. The originality and the main difference between SyMPA and other mobile agents platforms is the level of abstraction of the supported agents. Most other platforms (see section 7) offer Java APIs for designing mobile "agents" and all the needed mechanisms for management, deployment, communication, migration and security support. So, the programmer is actually writing Java *objects*. SyMPA offers all these mechanisms for intelligent and mobile agents described in a higher level language, CLAIM, agents that can invoke Java methods.

The next section resumes the characteristics of CLAIM; the third section illustrates its specifications, presenting a brief example of application implemented in CLAIM.The fourth section presents SyMPA's architecture and its three levels: the Central System, the Agent System and the Agent level; the fifth section is focused on the mobility in SyMPA, local and distant; the sixth section discuses the security and the fault tolerant mechanisms used for SyMPA. A comparison with the existing mobile agents platform, from the security point of view, is presented in the seventh section, followed by a conclusion.

2 Overview of CLAIM

CLAIM (Computational Language for Autonomous, Intelligent and Mobile agents) [5] is a declarative language situated at the intersection of two different domains, intelligent agents and process algebra, and tries to combine in a unified framework their main features and to compensate their disadvantages.

On one hand, the agent-oriented programming (AOP) languages, such as Agent-0 [21], AgentSpeak[24] or 3APL [11] allow representing the mental state of the agents, containing beliefs, goals, intentions or abilities, offer reasoning capabilities and communications primitives, but do not support agents' mobility. On the other hand, concurrent languages such as the ambient calculus [3], the safe ambients [16] or Klaim [4] allow representing concurrent processes, that can communicate and migrate in a distributed environment, some of them having well defined operational semantics, but in none of these languages it is possible to represent intelligent agents, with explicit believes, plans, goals or reasoning. Telescript [25] and April [17] are focused on the agents' mobility; nevertheless, these languages do not have neither the expressiveness and the reasoning capabilities of the agent oriented programming languages nor the formal solidity of the processes algebras.

CLAIM allows the design and the implementation of distributed MAS, located on several connected sites. On each site there is a hierarchy of agents. An agent in CLAIM is an autonomous, intelligent and mobile entity that has a list of local processes concurrently executed and a list of sub-agents. An agent has also an authority that is used for security reasons during the interactions between sites. In addition, an agent has mental components such as knowledge, capabilities and goals, that allow a reactive behavior (execute processes when messages arrive) or a goal driven behavior (execute processes in order to achieve goals). An agent can create new agents, can invoke methods implemented in other programming languages (e.g. Java methods, in this version), can communicate with other agents and can migrate using the mobility primitives inspired from the ambient calculus [3]. In CLAIM, agents and classes of agent can be defined using:

defineAgent *agentName* **{**
  authority=null; | *agentName* ***;***
  parent=null; | *agentName* ***;***
  knowledge=null; | **{** *(knowledge;)+***}**
  goals=null; | **{** *(goal;)+***}**
  messages=null; | **{** *(queueMessage;)+***}**
  capabilities=null; | **{** *(capability;)+***}**
  processes=null; | **{** *(process |)* process* **}**
  agents=null; | **{** *(agentName;)+***}**
}
defineAgentClass *className ((arg,)*)* **{...}**

The **knowledge** component represents the agent's information about the other agents (*i.e.* about theirs capabilities or their classes) or about the world.

Each agent has a set of **goals** that he is trying to achieve using his capabilities or asking capabilities' execution from other agents.

Each agent has a queue for storing the arrived **messages**. An agent can send asynchronous messages to itself or to another agent (*unicast*), to all the agents in a class (*multicast*), or to all the agents in the system (*broadcast*), using the primitive:

send(*receiver,message****)***

In CLAIM there are three types of messages:

1. *propositions*, defined by users and used to activate agents' capabilities;
2. the **messages concerning the knowledge**, used by agents to exchange pieces of information about their knowledge base and capabilities;
3. the **mobility messages**, used by the system in the mobility operations for asking, granting or not granting mobility permissions.

The **capabilities** component represents the actions an agent can do in order to achieve his goals or that he can offer to other agents. A capability triggers a process according to a message if a (optional) pre-condition is verified. It may have possible effects (a sort of post-conditions):

```
capability ::= capabilitySignature {
    message=null; | message;
    condition=null; | condition;
    do { process }
    effects=null; | { (effect;)+ }
}
```

A condition can be a Java function that returns a *boolean*, an achieved effect, a condition about agent's knowledge or sub-agents, or a logical formula.

Each agent can concurrently execute several process. A process can be a sequence of processes, an instruction (execute a process for all the agent's knowledge or sub-agents that satisfy a criteria), a variable's instantiation, a function defined in Java, the creation of a new agent, a mobility operation (detailed in Section 5) or a message sending:

```
process ::= process.process
          | forAllKnowledge(knowledge) { process }
          | forAllAgents(agentName) { process }
          | ?x=(value | Java(objName.function(args)))
          | Java(objectName.function(args))
          | newAgent agentName:className( (arg,)*)
          | open (agentName)
          | acid
          | in (mobilityArgument,agentName)
          | out (mobilityArgument,agentName)
          | moveTo (mobilityArgument,agentName)
          | send (receiver,message)
```

The previous elements allow two types of reasoning for the CLAIM agents, concurrently executed:

- *forward reasoning* (or reactive behavior): an agent activates capabilities when the corresponding messages arrive and the conditions are verified;
- *backward reasoning* (or goal-driven behavior): an agent executes capabilities in order to achieve his own goals.

The next section illustrates these specifications on an example, showing why we emphasize that SyMPA supports mobile agents rather than mobile code. At the language level, we are currently working on the operational semantics of CLAIM, that must take into account the mobility, the communication and the specificity and complexity of cognitive agents.

3 A Digital Library Application

One of the most important properties of CLAIM is the expressiveness. In order to compare it to other AOP languages, we easily translated in CLAIM several applications (without using mobile agents, not provided in these languages), such as: *Airline Reservation* from Shoham's AGENT-0 [21], a "*bolts making*" scenario from AgentSpeak [24], and FIPA-ACL protocols[1]. We also utilized CLAIM and SyMPA for programming an information research on the Web using mobile agents application [5], an electronic commerce application [22], the modelling of the coffee market in Verarcuz, Mexico and a load balancing and resource sharing application using mobile agents [14].

As the goal of this paper is to present the platform, we describe next only a brief part of a digital library application implemented in CLAIM using the SyMPA platform. There are several actors in our application: The Clients, the Searching Agents, created by Clients, the Librarians and the Section Librarians. A Client searches books satisfying some criteria; he creates a Searching Agent for each query and he migrates to a Library (with the created sub-agents). The Searching Agents migrate to Sections, borrow the books corresponding to their criteria, possibly asking for the Section Librarian's opinion and return to the Client. Finally, the Client leaves the Library. When there are to many clients on a site and the local resources are over-loaded, a Librarian can make migrate some of the sections to other computers less loaded. The figure 1 presents a sketch of the application.

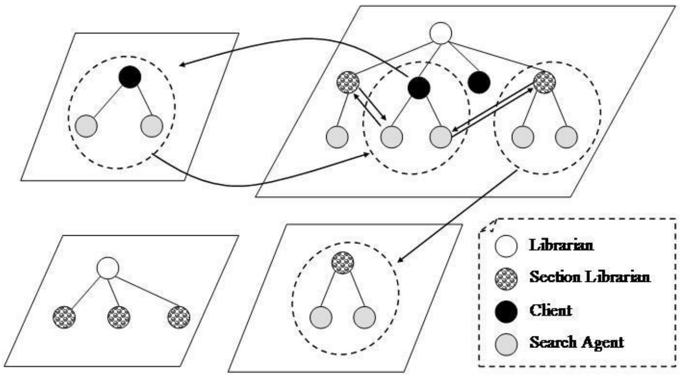


Fig. 1. Application's Architecture

In order to make the example easier to understand, only a part of the application's code in CLAIM is presented above. Thus, we can see two of the capabilities of a client, one for creating one Search agent for each document request and for migrating to the known library and the second for borrowing the document advised by the Search agents. We also can see the Searcher's capability for migrating to the corresponding section and for choosing a document after consulting the index and after asking the section librarian's opinion.

[1] www.fipa.org

```
defineAgent C {
  authority=Home;
  parent=Home;
  knowledge={search(SA1,Various,<>,<funny>);
    search(SA2,Fantastic,<Asimov>,<SF>);}
  goals=null;
  messages=null;
  capabilities={
   createSA() {
capability for creating the Search Agents and for going to the Library
    message=search();
    condition=null;
    do{forAllKnowledge(search(?ag,?s,?a,?w)){
     newAgent ?ag:SA(?s,?a,?w) }.
     moveTo(this,L).forAllAgents(?x:SA) {
      send(?x,search()) }
    }
    effects=null;    }
   borrowBook() {
capability for borrowing the books found by the SA
    message=borrow(?b);
    condition=null;
    do{ send(L,borrow(?b)). Java(C.borrow(this,?b)).
     send(this, goHome()) }
    effects=null;    }
   ...
  }
  processes=null;
  agents=null;
}

defineAgentClass SA(?s,?auth,?kw) {
  authority=null;
  parent=null;
  knowledge={search(?s,?auth,?kw);}
  goals=null;
  messages=null;
  capabilities={
   goToSection() {
capability for migrating to the interesting section
    message=search();
    condition=null;
    do{out(this,parent).in(this,?s).
     Java(SA.evaluateIndex(?auth,?kw)).
     Java(SA.askOpinion(?auth,?kw)).
     ?b=Java(SA.choose(?auth,?kw,this)).
```

```
      out(this,parent).in(this,authority).
      send(authority,borrow(?b)).acid }
    effects=null;
   }
  }
  processes=null;
  agents=null;
}
```

4 SyMPA's Architecture

SyMPA (French: Systéme Multi-Plateforme d'Agents) is a mobile agents platforms. The main advantage to other mobile agents platforms is that SyMPA supports agents implemented in CLAIM, an agent-oriented programming language while the other platforms (see section 7) support agents implemented using mainly object-oriented languages (e.g. Java in most cases). Nevertheless, a CLAIM agent deployed in SyMPA can use Java methods and our current work tackles the interoperability with other platforms. However, there are several requirements that must be satisfied by every solid mobile agents platform: support for agents' management, identification, communication, migration and security.

As mentioned before, SyMPA is compliant with the specifications of the MASIF [18] standard from OMG. MASIF provides a set of interfaces and definitions for the agents' management, identification, authentication, localization, tracking, communication, mobility and security. The main components of the MASIF architecture (figure 2) are:

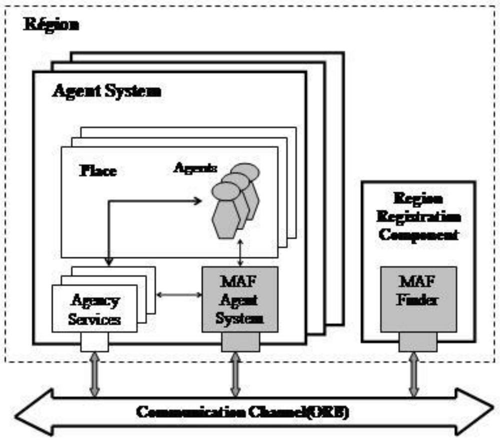


Fig. 2. MASIF's Architecture [18]

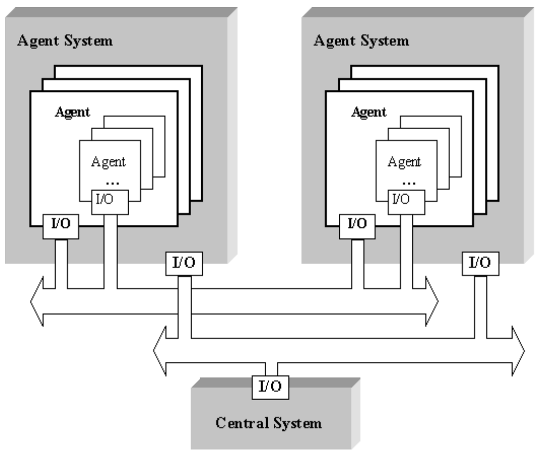


Fig. 3. SyMPA's Architecture

the agents belonging to an authority,

the agent systems - platforms that can create, interpret, execute, transfer and terminate agents,

the places - execution contexts and

the regions - sets of agent systems of the same authority that can be regarded as security domains.

There are two important interfaces, MAFAgentSystem, corresponding to each agent system and that provides the necessary services and MAFFinder, associated to a region and that offers functions for the localization of agent systems, of places and of agents in a region.

SyMPA is implemented using the Java language [12] and offers all the mechanisms needed for the design and the secure execution of a distributed MAS. SyMPA consists of a set of connected computers; on each computer there is an agent system (AS). There is also a Central System (CS) that has management functions. For a secure execution of a MAS designed in CLAIM, the platform also provides mechanisms for management, authentication, authorization, resources access control and fault tolerance. This architecture (fig. 3) has three levels that will be presented bellow.

4.1 The Central System

The CS provides services for agents' and ASs' management and localization. In MASIF, this corresponds to the MAFFinder interface. In SyMPA, the CS administrates (in this version) all the agents. In the future versions of SyMPA, the system administrator will be able to choose between different management solutions, in accordance with the current application.

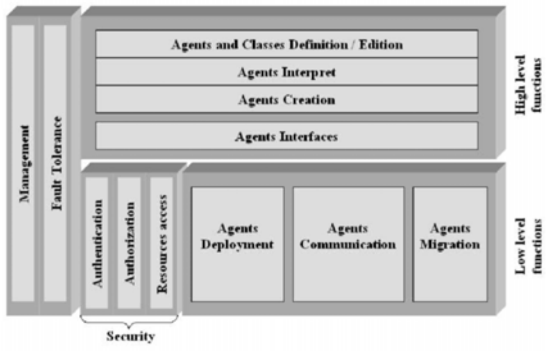


Fig. 4. SyMPA's features

4.2 The Agent System

An AS is deployed on each connected computer at the platform. It corresponds to the MASIF MAFAgentSystem interface. It provides (see fig. 4):

- high level mechanisms: a graphical interface (fig. 7) for defining CLAIM agents and classes, an interpret for verifying the definitions' syntax and interfaces for the running agents;
- low level mechanisms, for agents' deployment, communication, migration;
- management, fault tolerance and security mechanisms, such as the agents' authentication, authorization and resources access control;

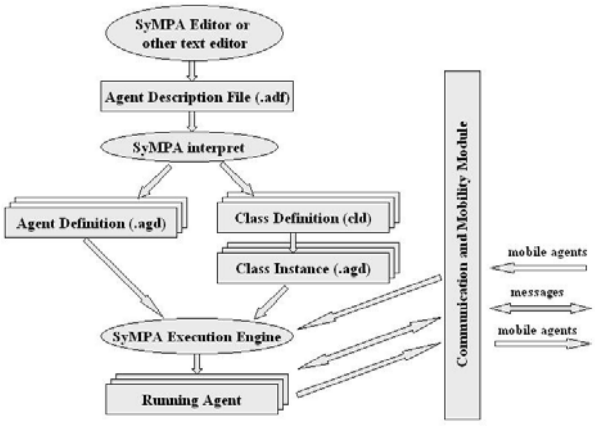


Fig. 5. Running agents in SyMPA

An AS offers en editor where the agents' designer can define the agents or the classes of agents needed for the application, in a .adf (agents description) file.

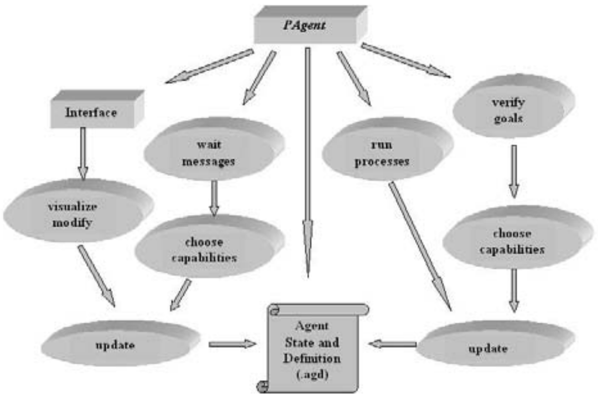
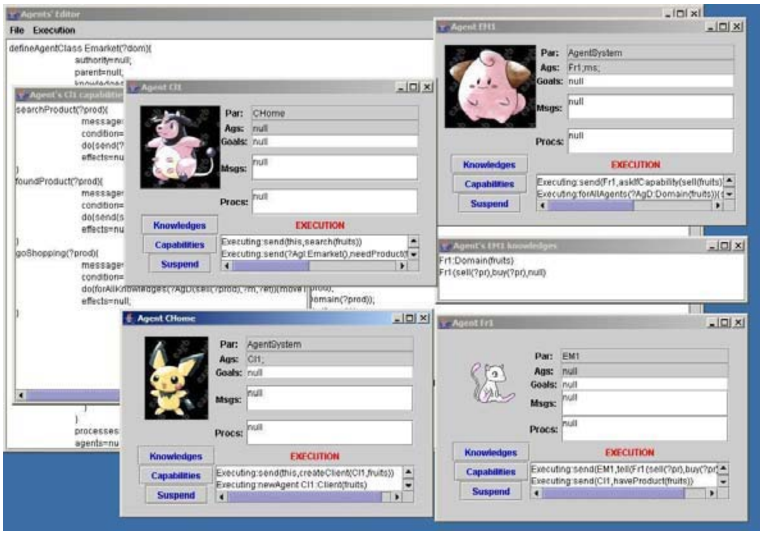


Fig. 6. A running agent's components

Nevertheless, any other text editor can be used. The definitions are then interpreted, the CLAIM syntax is verified and the agents (.agd) and classes (.cld) files are created in a format understandable by the Execution Engine (conf. figure 5). The compiler was implemented using JavaCC (Java Compiler Compiler) [13]. The running agents are charged in memory and executed. There is a corresponding (optional) graphical interface for each running agent, where one can visualize agent's behavior, communication and migration. The AS is also in charge with the communication with other ASs or the CS and with the mobility, taking into account the security constraints, that will be presented in the section 6.



```
defineAgentClass Emarket(?dom)(
                authority=null;
                parent=null;
```

```
searchProduct(?prod)(
                message
                condition=
                do(send(?
                effects=nu
)
foundProduct(?prod)(
                message
                condition=
                do(send(s
                effects=nu
)
goShopping(?prod)(
                message
                condition=
                do(forAllKnowledges(?AgD(sell(?prod),?m,?ef))(move
                effects=null;
)
```

```
omain(?prod));
```

```
processes
agents=nu
```

Agent Cl1 — Par: CHome; Ags: null; Goals: null; Msgs: null; Procs: null

```
Executing:send(this,search(fruits))
Executing:send(?Ag:Emarket(),needProduct(f
```

Agent EM1 — Par: AgentSystem; Ags: Fr1;ms; Goals: null; Msgs: null; Procs: null

```
Executing:send(Fr1,askIfCapability(sell(fruits))
Executing:forAllAgents(?AgD:Domain(fruits))(
```

```
Fr1:Domain(fruits)
Fr1(sell(?pr),buy(?pr),null)
```

Agent CHome — Par: AgentSystem; Ags: Cl1; Goals: null; Msgs: null; Procs: null

```
Executing:send(this,createClient(Cl1,fruits))
Executing:newAgent Cl1:Client(fruits)
```

Agent Fr1 — Par: EM1; Ags: null; Goals: null; Msgs: null; Procs: null

```
Executing:send(EM1,tell(Fr1(sell(?pr),buy(?pr
Executing:send(Cl1,haveProduct(fruits))
```

Fig. 7. SyMPA's graphical interface

4.3 The Agents

An agent in SyMPA is an autonomous, intelligent and mobile entity, defined using the CLAIM language. Each agent is uniquely identified in the system. Like the ambients in the ambient calculus, the agents form hierarchies. With this representation, a MASIF place becomes just an agent that has several sub-agents. For each agent there is a process called *PAgent* that provides methods for the agent's state management, represented in a shared-memory zone. Also, *PAgent* launches several processes (fig. 6):

– a graphical interface, utilized to visualize the agent's behavior, communication and mobility (fig. 7);
– a thread that concurrently executes the agent's processes;
– a process that performs the agent's forward reasoning, by listening the arriving messages, selecting the corresponding capabilities, verifying their conditions and updating the current processes;
– a process that performs the agent's backward reasoning, by trying to execute capabilities that allow to achieve his goals.

5 Mobility in SyMPA

As mentioned before, SyMPA is a set of connected computers. On each computer (site) there is an AS on which there are deployed several hierarchies of agents. With this representation, from the destination (during mobility) point of view, we can distinguish two types of migration : local and remote migration.

5.1 The Local Migration

The local migration is the migration inside a hierarchy, using the primitives inspired from the ambient calculus. These primitives can be classified in moving primitives and inheritance primitives. Using the first type of primitives, an agent moves with all his components in the local hierarchy. The local moving primitives used in CLAIM (introduced in section 2 and represented graphically in fig. 8) are:

– **in(*mobilityArgument,agentName*)** used by an agent to enter with all his sub-agents and processes another agent that must be in the same neighborhood (*i.e.* having the same parent) in order to successfully execute the operation.

– **out(*mobilityArgument,agentName*)** used by an agent to exit with all his components the current parent.

The inheritance primitives are also inspired from the ambient calculus, but are adapted to the specificity of intelligent agents. They allow a dynamic gathering of capabilities and enrichment of the knowledge bases. The two inheritance primitives are:

- **open(*agentName*)**, used by an agent to open the boundaries of one of his sub-agents. The running processes, the sub-agents, the capabilities and the elements in the knowledge base of the later are inherited by the former.
- **acid**, used by an agent to open his own boundaries, his components being inherited by his parent.

Even if we are using, in CLAIM, mobility primitives inspired from the ambient calculus, there are some important differences. First, in the ambient calculus, the only condition for the mobility operations is a structure condition (*e.g.* for the *enter* operation, the involved ambients must be on the same level in the hierarchy). In CLAIM, we kept this condition, but we added an asking / granting permission mechanism, for an advanced security and control. Secondly, we are using the *mobilityArgument* for specifying the mobility granularity. Unlike the ambient calculus, in CLAIM it is possible the migration of the agent himself, of a clone of the agent or of a process:

mobilityArgument = ***this*** | ***clone*** | *process*

5.2 The Distant Migration

The distant migration is the migration between hierarchies. Using the primitive ***moveTo*** *(mobilityArgument,agentName)*, an agent directly migrates to another agent, without verifying a hierarchical condition. Nevertheless, the asking entering permission mechanism is used. The figure 8 presents a graphical representation of the local (**in** and **out**) and the distant (**moveTo**) migration in SyMPA.

In order to assure the efficiency and the security of all the interactions, we

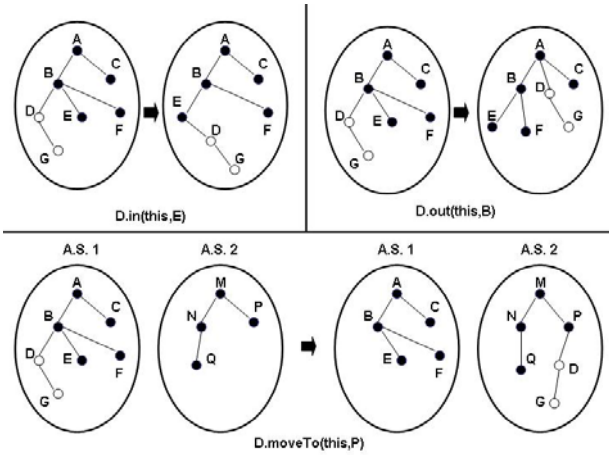


Fig. 8. Local and distant migration

proposed several protocols corresponding to all the agents' inter-site operations (communication and especially mobility), doubled by authentication and encryption protocols.

5.3 Weak Versus Strong Mobility

From the processes execution point of view, there are two kinds of mobility:

The Strong Mobility, where not only the code and the data is transferred but also the current state, the program counter and the execution stack. The running processes are resumed exactly where they were interrupted before the migration.

The Weak Mobility, where only the code and the data is transferred. The running processes are restarted at the destination. Java provides in the actual version facilities for the weak migration, such as the object serialization and Java RMI. There were approaches that modified the Java Virtual Machine (JVM) in order to support the strong object migration.

The mobility in CLAIM and SyMPA can be regarded at two levels.

First, there is a strong migration at the CLAIM level, because, before the migration, the state of an agent is saved and it is transferred to the destination. The agent's CLAIM processes are resumed from their interruption point. This is possible thanks to the way the mobility is done in SyMPA. An agent can be at any moment saved in a textual format, similar to the CLAIM definition, containing the current state (e.g. knowledge, messages, running processes). This representation is sent via network to the destination AS, in an encrypted format and the agent's execution is resumed from the saved state.

At the Java level, we use the *Java Virtual Machine* (JVM) migration facilities, so there is a weak migration. A Java method begun before the migration will be recalled after the arrival at the destination. Since the migration is achieved using the CLAIM primitives, unlike in other platforms, where there are Java objects that migrate during their execution, a solution can also be to let all the agent's running Java methods terminate before his migration.

5.4 The Implementation of the Mobility Operations

In this section we present intuitively how the mobility operations are implemented in the SyMPA platform.
On each site we have seen that there is an Agent System that offers mechanisms for agents' creation, management, communication and mobility by launching a process that we call *PSystem*. Also, for each agent, there is a corresponding process (called *PAgent*) that executes the agent (both the reactive and the pro-active behaviors) and offers a graphical interface. The operations that involve only the current agent are treated by the *PAgent* process. However, all the reconfiguring operations involve more than one agent and are managed by *PSystem*.

As we have already stated, all these operations can be classified in operations having as result a dynamic change in the system's structure and hierarchy (actually all the presented operations modify the hierarchies) and operations having as result a dynamic gathering of cognitive elements (in this category we have the inheritance operations: *open* and *acid*).

For the hierarchy change, if it is an operation requiring a structure condition (i.e. *in*, *out*, *open*) *PSystem* must verify first this condition. If the condition is verified, for the operations requiring permissions (the same three) the protocol for asking / giving permissions begins. Since the current structure cannot be changed during the execution of an operation, the involved agents are not allowed to treat other mobility messages. Thence *PSystem* updates the hierarchy and the MAS goes on with its execution.

Here is a pseudo-code of the operation $in(this, B)$ executed by an agent A:

```
Algorithm in {
  if neighbors(A,B){ // verify the structure condition
   A.blockMessagesTreatment(); // A blocks the mobility messages'
treatment
   A.askInPermission(B); // A asks enter permission from B
   if(A.receiveInPermission(B)) { // If B gives the permission,
    B.blockMessagesTreatment();//B blocks the mobility messages' treatment
    A.parent ← B; // B becomes the parent of A
    B.addAgent(A); // A is added in B's list of sub-agents
    A.resume();//the two agents unblock the mobility messages' treatment
    B.resume(); // and resume their execution
   } else {A.postPoneProcess(in); A.resume(); } // If A does not receive
    // the permission, he unblocks the mobility messages' treatment,
    // postpones the in operation and resumes the execution
  } else {A.postPoneProcess(in);} // A postpones the in operation
}
```

For the inheritance operations there are both a change in the agents' structure and a change in the internal state of the agents. For *open* (in the pseudo-code bellow, A is opening B), there is first a structure condition to be verified, followed by a permission asking. If these requirements hold, *PSystem* updates the hierarchy, the intelligent aspects are inherited by the parent agent and the MAS resumes its execution.

```
Algorithm open {
  if A.hasAgent(B){ // verify the structure condition
   A.blockMessagesTreatment();//A blocks the mobility messages' treatment
   A.askOpenPermission(B); // A asks open permission from B
   if(A.receiveOpenPermission(B)) {// If B gives the permission,
    B.blockMessagesTreatment();//B blocks the mobility messages' treatment
    forAll(ag ∈ B.agents) {A.addAgent(ag)}
    forAll(p ∈ B.processes) {A.addProcess(p)}
    forAll(k ∈ B.knowledge) {A.addKnowledge(k)}
    forAll(c ∈ B.capabilities) {A.addCapability(c)}
    //the agents, processes, knowledge and capabilities are inherited by A
    stop(B); // B stops his execution and disappears from the MMAS
    A.removeAgent(B); // B is eliminated from the list of A's sub-agents
    A.resume();//A unblocks the messages' treatment and resumes his execution
    } else {A.postPoneProcess(open); A.resume();} If A does not receive
      // the permission, he unblocks the mobility messages' treatment,
      // postpones the open operation and resumes the execution
  } else {A.postPoneProcess(open);} // A postpones the open operation
}
```

6 Security in SyMPA

6.1 Attacks and Solutions

The mobile agents are programs running in a distributed and insecure environment (e.g. the Internet) where there are possible two main types of attacks [9] that are resumed bellow, together with some of the existing solutions:

1. Attacks **against the host** AS: a mobile agent can damage local resources, configurations or files, can overload some host resources or services, can access very important, private information, can have a annoying behavior for the users of the host system, can execute actions for which he does not have the rights, in the name of a higher authority or can, under an apparent inoffensive behavior, start some damaging application triggered by a specific event. The main security techniques used in order to prevent these kind of attacks are:

The Authentication: the utilization of digital signatures or certificates; nevertheless, this technique does not guarantee that the mobile agent will be harmless;
Resources Access Control: the management and the control of the agent's access to the system's resources (e.g. the rights to access files, to use the network, the memory, the CPU utilization, etc), in function to the authorization level granted to the agent and of the security policies of the host system;
Code Verification: to find damaging, illegal or not allowed instruction in the code of an agent; however, this does not guarantee a safe execution of the agent;
Limitation Techniques: the limitation of the execution time of an agent, of the number of clones or of the number of destinations;
Audit: the record of all the agent's activities in all its execution stages in order to detect possible incorrect actions.

2. Attacks **Against Mobile Agents**, during the migration and during the execution on a host system. The attacker can be, during the migration, a third person that can monitor the network traffic and can access or even damage important private data of the agent, and during the execution, another agent or the host. This last type of attack is the most difficult to prevent. A host can destroy the agent, damage the agent's data or access important and secret data, can not be capable to assure or refuse the access to a resource (denial of service), can delay his execution or can give to an agent misinformation (e.g. about the next destination). Two important types of techniques are used for protecting the agents:

Encryption Techniques: the encryption of the agents during the migration, of the exchanged messages, the encryption of the agent's code or of his private data

during the execution, in order to not be accessed by malicious hosts. However, the code encryption can be dangerous for the host systems. Two main types of encryption are usually utilized, public key or secret key encryption;
Fault Tolerance Techniques: the replication of an agent, to ensure that a mobile agent arrives at his destination, the persistence, to protect an agent against a host failure and the redirection, to change the trajectory of an agent in order to avoid damaged hosts.

6.2 SyMPA's Security Elements

The security element in SyMPA takes into account the two types of attacks in the mobile agents systems. There are also used fault tolerance techniques for the platform. We are using existing common techniques, but as we will see, we are covering all the important aspects.

Protecting Host Machines

1. **Authentication:** each agent is uniquely identified in the system. An agent acts in the name of an authority that can be an organization, the agent that created him or the authority of the agent that created him. When created, an agent is registered to the CS, with his name, the name of the authority and the agent's public key.
2. **Resource Access Control:** Each AS has a set of permissions for agents belonging to known authorities. It verifies the authority of an incoming agent, at the CS. For agents belonging to unknown authorities, it will give minimum permissions. The rights for a CLAIM agent can be analyzed at two levels:

– the CLAIM level: corresponds to the resources accessed by a CLAIM process. These permissions include the number of sub agents an agent is allow to create, the permission to communicate with other local or remote agents, the mobility permissions, the time of execution and the permission to invoke Java methods (and the number of threads he can create). When an AS detects that an agent is trying to execute a CLAIM process for which he does not have the rights, first it will send a CLAIM specific message, requesting the immediate depart, message that will have a by-default treatment consisting in a migration to the authority site, and if after a limited time the agent is still on the local host, the AS will destroy him.
– the Java level: corresponds to the Java methods called by an agent and is based on Java security mechanisms. The permissions concern the reading/writing from/on the disk (and from/in what directory), and the network access. Using the Java security mechanisms, an agent will not be capable to execute forbidden actions; security exceptions will be thrown.

3. **Audit:** In SyMPA there are monitored only the critical activities, because of the delays introduced by the monitoring. Therefore, the AS monitors the external communication, the distant migration, the creation of an agent and the called Java methods. However, the audit does not prevent an agent to do damaging actions, but can be used after to identify the malicious agents, their authority and

to change in accordance with this the future rights and treatment of the same agent or of agents belonging to the same authority. In this version, the audit files have a user-friendly format and are analyzed by system's administrators. In a future version, these files will be interpreted using some automated mechanisms.

Protecting Agents

1. **Cryptography:** The public and secret key cryptography is used at the three SyMPA levels and for all the exchanged messages or mobile agents. Each platform entity, including the CS, the ASs and the agents, has a public key registered to the CS. A cryptographic protocol is added at each message exchange. When an entity tries to communicate with another one, he will initiate the protocol using the public key of the later, that will create a secret session key used during the messages exchange. So all the messages, including the mobile agents on their route, are encrypted.

For fault tolerance reasons, the agents are stored on the disk with all theirs components. To protect a stored agent from other agents, the agent's definition on the disk is also encrypted by the AS.

However, we cannot prevent in SyMPA attacks from the host systems against the running agents. In order to execute an agent, the platform must access the agent's CLAIM code and important data. Because a CLAIM agents can have several behaviors in function of his knowledge, goals and capabilities, the SyMPA execution engine must have full access at the agent's components. Our SyMPA's implementation is a safe one and the agents are correctly and safely executed. The only possibility to have attacks from a host against an agent is to eventually modify the AS of the SyMPA implementation. But all the ASs must registry to the CS, so a malicious AS will not be very difficult to detect.

Fault tolerance mechanisms are also utilized at the SyMPA's three levels. For the CS is used a replication technique. There are several running CSs, with different priorities, from which only one is active. This one sends periodically to the others the current configuration. When the inactive CSs do not receive the configuration in time, they will try to contact the active CS. If it does not respond, a message will circulate between them, where every CS will write its priority. After visiting all the CSs, the message will go the CS with the best priority, this one will become the active CS, resuming the execution from the last received configuration. For the ASs, all its data is periodically saved on disk, with the current state of all the running agents, in order to resume the systems execution after a breakdown. The agents are also regularly stored on disk by the AS, in an encrypted format. Before an agent's migration, he is stored on the disk and after the confirmation of the successful arrival at the destination, the agent is erased from the disk. If the arrival confirmation is not received, the AS will try again to send the stored agent. If the case of a breakdown of the destination system the agent is either kept on the disk until the destination is recovered, either sent to his authority.

7 Related Work

In this section we will resume the characteristics and the security elements of other platforms supporting mobile code. Let's note that a state of art of the existing agent oriented programming languages and of the concurrent languages for programming mobile processes can be found in [6].
Aglets[1] are mobile and communicants Java objects, developed by IBM. The main security elements provided are the authentication of the owner, the designer and the sender of a mobile agent, the data encryption and the authorizations for resources access, based on Java specifications.
Ara[19] is a platform that supports mobile agents implemented in Java, TCL or C/C++. The security elements are reported at the three entities of Ara, the designer of an agent, the owner of an agent and a host system, and include: the authentication, a vector of resources access rights, some fault tolerance mechanisms and the SSL protocol for secure data transmission.
Ajanata[8] is a Java based mobile agents platform, with a weak mobility implemented using the Java's serialization facilities. There are host protection mechanisms (agents' credentials, a list of resources access rights, security manager, authenticated communication, a mobility protocol similar with those of SyMPA) and agent protection elements (cryptography, audit, agents' credential protection, restrictive access to an agent's registered data).
Concordia [23] is a Java platform developed by Mitsubishi. The mobility uses the serialization facilities. Between the security elements we can note the users' identification, the permissions for resources access, based on Java, the SSL protocol for data transmission, the data encryption, using Java cryptography and fault tolerance mechanisms.
D'Agents [7] is a platform for mobile agents that can be implemented in the Tcl, Java or Scheme languages. The main security elements are the authentication, the authorization, the resources access control (an agent does not directly access a resource, but through resource management agents), the data encryption and digital signatures.
Grasshopper[10] is MASIF compliant and uses for guaranteeing the security the authentication (X.590 certificates), the confidentiality (SSL protocol for data encryption) the message integrity (MCA codes), the resources access control (based on Java) and fault tolerance mechanisms.

Other mobile agents platforms have been developed, such as Bee-gent, Gypsy, Hive, James, MARS, MOLE, Odyssey, SOMA, Voyager, most of them supporting Java agents, but all of them trying to ensure a minimum level of security, a vital aspect in the mobile agents applications. The table above presents a summary of security features in the studied platforms (some aspects were not clear deduced from the documentation) and in SyMPA. The agents' "*after*" protection against hosts means that a malicious host will be detected after it *attacked* an agent. The attack cannot be undone and the agent can be damaged, but the host will not be trusted in the future.

	Host protection			Agent protection			
	authentication	resources access	audit	against agents	persistence	encryption	against hosts
Aglets	Yes	Yes	?	?	No	Yes	No
Ara	Yes	Yes	?	?	Yes	Yes	?
Ajanta	Yes	Yes	Yes	Yes	?	Yes	No
Concordia	Yes	Yes	?	Yes	Yes	Yes	After
D'Agents	Yes	Yes	?	?	?	Yes	No
Grasshopper	Yes	Yes	Yes	Yes	Yes	Yes	No
Gypsy	Yes	Yes	?	?	?	?	No
Hive	Yes	Yes	?	?	?	?	No
James	?	Yes	?	?	Yes	?	No
Mole	Yes	Yes	?	?	Yes	No	No
Odyssey	Yes	Yes	No	?	No	No	No
SOMA	Yes	Yes	?	?	?	Yes	After
Voyager	Yes	Yes	?	?	?	Yes	?
SyMPA	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**After**

8 Conclusions and Future Work

We presented in this paper a mobile agents platform called SyMPA, that supports mobile agents implemented using a new agent oriented programming language - CLAIM, that homogeneously combines elements form the agent oriented programming languages (for representing agents' intelligence and communication) with elements from the concurrent languages (e.g. the ambient calculus) for representing agents' mobility. SyMPA offers all the necessary mechanisms for the design and secure execution of distributed MAS written in CLAIM. Even if there are many mobile agents platforms, their impact and utilization in real live application is not at the level it should be. It is a common opinion that the mobile agents technology is very useful in the development of distributed applications, but the main reasons for its reduced utilization are the **security** problems, of vital importance in a distributed and unsafe environment like the Internet and the **interoperability** between different platforms.

An important part of this paper was dedicated to the security aspects in the mobile agents world. However, the reached security level is not as high at it should be in order to have really safe distributed applications on the Internet and more significant steps need to be done in this direction; a part of our current work is focused on these aspects.

From the interoperability point of view, SyMPA is compliant with the specifications of the MASIF standard from OMG, a collection of interfaces and definitions that tries to standardize the aspects of a mobile agent platform. Nevertheless, this standard is used by few of the existing platforms (only Grasshopper and SOMA from the studied platforms). In SyMPA we are working on having interoperability with agents coming from different platforms and implemented in other programming languages. For this purpose we are currently developing a Web Services approach, where the agents publish their capabilities as Web

Services that can be found and invoked by other agents (CLAIM or from other platforms) or even by independent applications. Nevertheless, a real interoperability between heterogenous mobile agents is difficult and must take into account interaction protocols, language and ontology problems.

References

1. Aglets Workbench: http://www.trl.ibm.co.jp/aglets
2. Cardelli L.: Mobile Ambients Synchronization. SRC Technical Note, Digital Equipment Corporation System Research Center, (1997)
3. Cardelli L., Gordon A.D.: Mobile Ambients. Foundations of Software Science and Computational Structures, Maurice Nivat (Ed.), LNCS, **1378**, (1998) 140-155
4. deNicola R., Ferrari G.L., Pugliese R.: Klaim: a Kernel Language for Agents Interaction and Mobility". IEEE Transactions on Software Engineering (1998) 315-330
5. El Fallah-Seghrouchni A., Suna A.: An Unified Framework for Programming Autonomous, Intelligent and Mobile Agents. Proceedings of CEEMAS'03, LNAI, **2691** (2003) 353-362
6. El Fallah-Seghrouchni A., Suna A.: CLAIM: A Computational Language for Autonomous, Intelligent and Mobile Agents. Proceedings of ProMAS'03, LNAI, **3067** (2003) 90-110
7. Gray R.S., Kotz D., Cybenko G., Rus D.: D'Agents: Security in a multiple-language, mobile-agent system. Mobile Agents and Security, LNCS, **1419** (1998) 154-187
8. Karnik N.M., Tripathi A.R.: Security in the Ajanta Mobile Agent System. Software: Practice and Experience **31(4)** (2001) 301-329
9. Greenberg M.S., Buyington J.C., Harper D.G.: Mobile Agents and Security. IEEE Comunications Magazine, (1998) 76-85
10. Grasshopper on-line at http://www.grasshopper.de
11. Hindriks K.V.,deBoer F.S., van der Hoek W., Meyer J.J.Ch.: Agent Programming in 3APL. Intelligent Agents and Multi-Agent Systems, **2** (1999) 357-401
12. Java on-line at http://java.sun.com
13. JavaCC on-line at https://javacc.dev.java.net/
14. Klein, G., Suna A., El Fallah-Seghrouchni A.: Resource sharing and load balancing based on agent mobility. Proceedings of ICEIS '04 (2004)
15. Lange D.B., Oshima M: Seven Good Reasons for Mobile Agents. Communications of the ACM **42** (1999) 88-89
16. Levi F., Sangiori D.: Controlling Interference in Ambients. Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (2000) 352-364
17. McCabe F.G., Clark K.L.: April Agent PRocess Interaction Language. Intelligent Agents: Theories, Architectures, and Languages, LNAI, **890** (1994)
18. Milojicic D., Breugst M., Busse I., Campbell J., Covaci S., Friedman B., Kosaka K., Lange D., Ono K., Oshima M., Tham C., Virdhagriswaran S., White J.: MASIF, The OMG Mobile Agent System Interoperability Facility. Proceedings of Mobile Agents, LNAI, **1477** (1998) 50-67
19. Peine H.: Security Concepts and Implementation in the Ara Mobile Agent System. Proceedings of the Seventh IEEE Workshop on Enabling Technologies: Infrastructure for the Collaborative Enterprises, Stanford University, USA, (1998)
20. Rao A.S., Georgeff M.P.: BDI Agents: From Theory to Practice. Proceedings of the First Intl. Conference on Multiagent Systems (1995)

21. Shoham Yoav: Agent Oriented Programming. Artifficial Intelligence, **60** (1993) 51-92
22. Suna A., El Fallah-Seghrouchni A: Programming e-commerce applications using CLAIM. Proceedings of CSCS14, Bucharest (2003)
23. Walsh T., Paciorek N., Wong D.: Security and Reliability in Concordia. Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences Proceedings of HICSS'98, **7** (1998) 44-53
24. Weerasooriya D., Rao Anand S., Ramamohanarao K.: Design of a Concurrent Agent-Oriented Language. Intelligent Agents. Proceedings of First International Workshop on Agent Teories, Architectures and Languages (ATAL'94), LNAI, **890** (1994)
25. White J.: Mobile agents. Software Agents, Bradshaw, J. Ed., MIT Press, (1997)

Bridging the Gap Between AUML and Implementation Using IOM/T

Takou Doi[1], Nobukazu Yoshioka[2], Yasuyuki Tahara[2], and Shinichi Honiden[1,2]

[1] University of Tokyo, 7-3-1, Hongo, Bunkyo-ku, Tokyo, Japan
[2] National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, Japan
{tdoi, nobukazu, tahara, honiden}@nii.ac.jp

Abstract. Multi-agent systems are attractive means for developing complex software systems. However, multi-agent systems themselves tend to be complex, and certain difficulties exist in developing them. One of the difficulties is the gap between design and implementation especially for interaction protocols. In this paper, we propose a new interaction protocol description language called IOM/T. Interaction protocols described using IOM/T have clear correspondence with AUML sequence diagrams and the description can be consolidated into a single unit of IOM/T code. Then, we show how the process of implementing Java-based agent-platform code from AUML design can be carried out, and how IOM/T effectively bridges the gap between design and implementation.

1 Introduction

Multi-agent systems are attractive means for developing complex software systems. However, multi-agent systems themselves tend to be complex, and certain difficulties exist in developing them. One of the difficulties is the gap between design and implementation [3], [1]. Especially the gap about interaction protocols is large. In Analysis and Design phase, developers use an agent-oriented method such as Gaia[16]. In this phase, interaction protocols are considered as important as agents themselves. In the Implementation phase, however, developers use object oriented languages such as Java, and implement interaction protocols as agent capabilities. As a result, the actual implementation of interaction protocols becomes dispersed, and the correspondence between the messages transition and the message reception becomes unclear. Yet, interaction protocols of multi-agent system are quite complex, thus leading to problems such as the following:

1. Understanding the interaction protocols from code is difficult.
2. Maintenance of interaction protocols is difficult.

In this paper, we propose a new interaction protocol description language called IOM/T(Interaction Oriented Model by Textural representation). Using this language, we can separate interaction protocol code from agent code. Furthermore, we can implement an interaction protocol in a single unit of code. We also describe the development process using IOM/T and tools. During the process, we analyze and design multi-agent system on the basis of the interaction protocols. Then, we use a tool to generate a skeleton code of IOM/T for each interaction protocols. We refine the code by adding information of the implementation. Finally, we use a preprocessor to convert the IOM/T codes and the agents' codes into the Java codes for a certain agent platform. IOM/T and this development process will bridge the gap between the design and the implementation.

Below, this paper is structured as follows. In section 2, we propose IOM/T. Then, in section 3, we show the interaction based development process of multi-agent system. In section 4, we evaluate IOM/T by comparing it to an existing agent-platform. Related work is discussed in section 5, and some conclusions are presented in section 6.

2 IOM/T: Interaction Oriented Model by Textural Representation

In this section we present IOM/T and describe its syntactic rules.

2.1 Concepts

We think one of the factors which make the development of multi-agent systems difficult is that we can not deal with interactions in Implementation as same as in Design. We need a representation which holds information about both design and implementation. Furthermore interaction protocols are generic rules for communicating and do not depend on agents mental states. In other words, we can separate interaction code from agent code. This simplifies the representations of the interactions. Therefore we designed IOM/T on the basis of the following concepts:

1. Code describing the interaction protocols should have clear correspondence with the design.
2. Developers should be able to easily understand the interaction protocol flow.
3. Separation of Concerns should be achieved.
4. Interaction protocols should be allowed to be represented with a single unit of code.
5. Notation is based on Java.

2.2 Protocol Example

In the remainder of this paper, we will exploit a simple example to describe our language and methodology. We define the Iterated Ping protocol using an AUML[7] sequence diagram[8] as shown in Fig.1. Two roles, Sender and Receiver, participate in this protocol. The Sender sends a "ping" message to the Receiver, and the Receiver replies to that message. This sequence is repeated as long as the Sender wants to. The protocol is described using IOM/T as shown in Fig.2. This paper also includes an example code for the English Auction. A.1 includes the protocol representation in AUML sequence diagrams and A.2 includes the corresponding IOM/T code.

2.3 Definition of Protocols

In IOM/T a protocol is represented as a structure using the keyword *protocol*. The protocol structure has a unique identifier, and consists of definitions of roles and a interaction flow. Fig.2 defines an interaction protocol named PingProtocol.

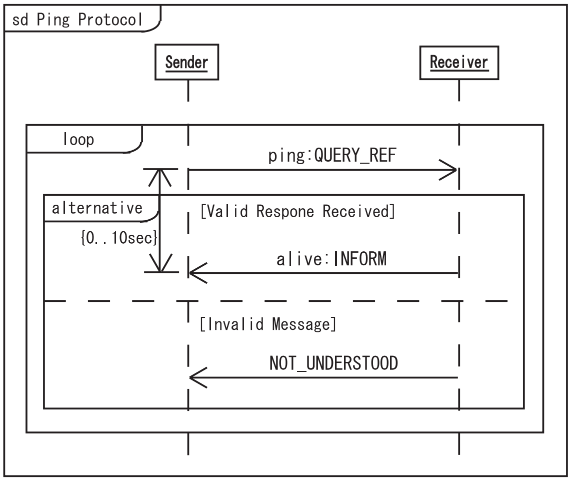


Fig. 1. Iterated Ping protocol represented using AUML sequence diagram

```
public protocol PingProtocol {
  player Sender {
    AID getTarget();
    isContinue();
    void knowAsDead();
    Date sendTime;
  }
  player Receiver {
  }
  interaction {
    while (Sender.isContinue()) {
      player Sender {
        ACLMessage ping = new ACLMessage();
        ping.setReceiver(getTarget());
        ping.setContent("(ping)");
        ping.setPerformative("QUERY_REF");
        sendAsync(ping);
        sendTime =
          Calender.getInstance().getTime();
      }
      player Receiver {
        ACLMessage ping = recvBlock();
        ACLMessage res = ping.createResponse();
        if (ping.getContent.equals("(ping)")) {
          res.setContent("(alive)");
          res.setPerformative("INFORM");
        } else {
          res.setContent(ping.getContent());
          res.setPerformative("NOT_UNDERSTOOD");
        }
        sendAsync(res);
      }
      player Sender() {
        while (true) {
          ACLMessage res =
            recvNonBlock();
          if (res != null &&
              res.getPerformative().equals("INFORM") &&
              res.getContent().equals("(alive)")) {
            break;
          }
          Date current =
            Calender.getInstance().getTime();
          if (current.getTime()
                    - sendTime.getTime()
                        >= 10 * 1000)) {
          knowAsDead();
          break;
        }
      }
    }
    player Sender {
      ACLMessage msg = new ACLMessage();
      msg.setReceiver(getTarget());
      msg.setContent("(end-of-loop)");
      msg.setPerformative("INFORM");
      sendAsync(msg);
    }
  }
}
```

Fig. 2. Iterated Ping protocol described in IOM/T

The syntactic rule of protocol structures is as follows:

$$\begin{array}{lll}
ProtocolDefinition & ::= & \text{protocol } Identifier\ ProtocolBody \\
ProtocolBody & ::= & \{\ ProtocolBodyDefinition\ \} \\
ProtocolBodyDefinition & ::= & PlayerDefinitions\ InteractionDefinition \\
PlayerDefinitions & ::= & PlayerDefinition \\
 & | & PlayerDefinitions\ PlayerDefinition
\end{array}$$

2.4 Definitions of Roles

A role is defined as a structure using the keyword *player*. The symbol $*$ after the keyword *player* represents the existence of several agents playing the role. A player structure has a unique identifier, and consists of definitions of sub-protocol, definitions of role functionality, and definitions of role information. When we have to deal with a large interaction protocol, we divide it into some sub-protocol and represent the whole interaction protocol by using them. Definition of sub-protocol specifies which sub-interaction the role participates in and what role it plays in the sub-protocol using the keyword *playing*. Definition of role functionality is specified using Java method definition notation. Definition of role information is specified using Java field definition notation. In Fig.2 the two roles, Sender and Receiver, are defined on lines 2 - 9. The code shows that the Sender has three functionalities and one variable for information. The functionality $getTarget()$ determines the target agent. The functionality $isContinue()$ is used to determine the end of loop. The functionality $knowAsDead()$ notifies the Sender agent that the target agent did not reply to the message. The information $sendTime$ is used to hold the time the Sender sent the message. On the other hand, the Receiver does not have any functionality and information.

The syntactic rule of player structures is as follows:

$$\begin{array}{lll}
PlayerDefinition & ::= & \text{player } *_{opt}\ Identifier\ PlayerBody \\
PlayerBody & ::= & \{\ PlayerBodyDefinitions_{opt}\ \} \\
PlayerBodyDefinitions & ::= & PlayerBodyDefinition \\
 & | & PlayerBodyDefinitions\ PlayerBodyDefinition \\
PlayerBodyDefinition & ::= & UseProtocolDefinition \\
 & | & AgentFunctionDefinition \\
 & | & InformationDefinition \\
UseProtocolDefinition & ::= & \text{playing } ProtocolName.PlayerName; \\
AgentFunctionDefinition & ::= & ReturnType\ FunctionIdentifier(Arguments_{opt}); \\
InformationDefinition & ::= & InformationType\ InfomationIdentifier;
\end{array}$$

2.5 Definition of Interaction Protocol Flows

An interaction protocol flow is defined as an structure using the keyword *interaction*. An interaction structure consists of a combination of sequential flow and repeated flow. The sequence of the role actions represents the sequential flow, and “while” structure represents the repeated flow as in Java notation. There are restrictions on this structure because of the feasibility for parallel process. One is that the “while” condition predicate must describe a single functionality of a single role. The other is that the role which

```
public class ACLMessge {
    public void setSender(AID sender);
    public AID getSender(AID sender);
     ...
    public void setPerformative(String performative);
    public String getPerformative();
     ...
    ACLMessage createResponse();
}
```

Fig. 3. Outline of the ACLMessage class

determines the end of loop must send notification messages at the end of loop to other roles. A role action is defined as a block labeled with the role identifier. This block contains a role action defined using Java and some language enhancement, namely expressions for using role functionalities and information, expressions for dealing with FIPA-ACL message, and expressions for controlling protocols. In Fig.2 the Sender's action is described on lines 12 - 20, the Receiver's action on lines 21 - 32 and the Sender's action on lines 33 - 49 represent a sequential flow, and this sequential flow is structured as a repeated flow on lines 11 - 50. The Sender sends notification for the end of loop in the Sender's action on lines 52 - 58.

The syntactic rules for defining an interaction flow are as follows:

$$\begin{array}{lll}
InteractionDefinition & ::= & \text{interaction } InteractionBody \\
InteractionBody & ::= & \{\ InteractionBlocks\ \} \\
InteractionBlocks & ::= & InteractionBlock \\
 & \mid & InteractionBlocks\ InteractionBlock \\
 & \mid & \text{while(} WhilePredicate \text{) } \{\ InteractionBlocks\ \} \\
InteractionBlock & ::= & \text{player } PlayerIdentifier\ \{\ ActionBlock\ \} \\
WhilePredicate & ::= & PlayerIdentifier.FunctionIdentifier(Args_{opt}) \\
 & \mid & \text{(PlayerIdentifier must specify a single role)}
\end{array}$$

Expressions for Using Role Functionalities and Information. The use of a role functionality is represented as a method invocation in Java, and the use of a role information is represented as a field access in Java. In Fig.2 the functionality of Sender is used on line 11, 14 and 47. The information of Sender is used on line 18 and 45.

Expressions for Dealing with FIPA-ACL Message. In order to deal with FIPA-ACL messages, the following classes and functions are used:

1. AID : This class represents the agent identifier.
2. ACLMessage : This class represents FIPA-ACL messages and its outline is laid out on Fig3. It has getter/setter methods to handle the terms specified in the messages. This class also has a method to create a reply message as an instance of ACLMessage class.
3. send/receive functions : These functions are used to send or receive an instances of ACLMessage class. $sendSync$ function represents synchronous transmission, and $sendAsync$ function represents asynchronous transmission. $recvBlock$ function represents a blocking reception, and $recvNonBlock$ function represents a non-blocking reception.

```
public class MyAgent extends Agent {
    playing PingProtocol.Sender {
        getTarget = getPingAgent;
        isContinue = isPingContine;
        knowAsDead = knowAsDead;
    }
    AID getPingAgent() { ... }
    boolean isPingContinue() { ... }
    void knowAsDead() { ... }
}
```

Fig. 4. Agent code example using the Iterated Ping protocol

Expressions for Controlling Protocols. The following functions are used to control the protocols:

1. Begin sub-protocol : The function $beginProtocol$ is used to begin a sub-protocol. The next sentence means the beginning of a sub-protocol specified by the identifier $ProtocolIdentifier$ as the role specified by the identifier $RoleIdentifier$.

$$beginProtocol(ProtocolIdentifier.RoleIdentifier)$$

2. Terminate current protocol : The function $terminateProtocol$ is used to terminate the current protocol.

2.6 Using Interaction Protocols

Above, we show the description of interactions. We show the description of agents with a focus on the use of interaction protocols. The description of the other aspects are out of the scope of this paper. In order to use interaction protocols, functionality mappings are defined in the agent code. A functionality mapping represents how methods of each role functionality are implemented using the keyword $playing$. The function $beginProtocol$ is used to begin a sub-protocol. Fig.4 shows an example of using the Iterated Ping protocol.

The syntactic rules for mapping functionality are as follows:

$PlayingDefinition$	::= playing $ProtocolIdentifier.PlayerIdentifier$
	$\mid\ PlayingDefinitionBody$
$PlayingDefinitionBody$	::= $\{FunctionMappings_{opt}\}$
$FunctionMappings$	::= $FunctionMapping$
	$\mid\ FunctionMappings\ FunctionMapping$
$FunctionMapping$	::= $NormalFunctionMapping$
	$\mid\ ProtocollFunctionMapping$
$NormalFunctionMapping$	::= $FunctionIdentifier\ =\ MethodIdentifier$
$ProtocolFunctionMapping$	::= $ProtocolSpecifiers.FunctionIdentifier$
	$\mid\ =\ MethodIdentifier$
$ProtocolSpecifiers$	::= $ProtocolSpecifier$
	$\mid\ ProtocolSpecifiers\ ProtocolSpecifier$
$ProtocolSpecifier$	::= $ProtocolIdentifier.PlayerIdentifier$

2.7 AUML and IOM/T

The current version of IOM/T can not fully represent all the interactions that can be described with AUML sequence diagrams. An AUML sequence diagram corresponds to an protocol structure and a time-line corresponds to a role. However, IOM/T can not represent every kind of CombinedFragment[8]. Only CombinedFragments whose interaction operators are “Loop” can be represented by “while” structure. We have to describe the meaning of the other interaction operators in role actions. Although we can extend the language adding new structures to deal with these interaction operators, it is hard to generate codes which them such as alternative, option, parallel and so forth. This extension is included in our future work. In fact, we aim at providing the semantics and proving the equivalence with AUML sequence diagrams.

3 Development Process

In this section we provide an overview of our development process based on interaction protocols. The process is roughly divided into two phases. In the Analysis and Design phase, developers extract requirements and functionalities, and design the interaction protocols and agents in the system. Then, in the Implementation phase, developers implement the interaction protocols and agents on the basis of on the design decisions in the previous phase. This phase includes testing. We show the development process focusing on the Implementation phase due to limitations of space.

3.1 Analysis and Design

The first phase is the Analysis and Design phase. In this phase we extract the system requirements, and decide how these requirements are realized.

The products of this phase are the design of interaction protocols and agents in the system. Each interaction protocol consists of roles and message sequence. A role contains functionalities to fulfill the interaction. A message sequence is represented using an AUML sequence diagram. Each agent is endowed with roles that it has to play.

3.2 Implementation

The second phase is the Implementation. We implement interactions using IOM/T and agents using APIs that depend on the agent-platform. In this paper we concentrate on the implementation of interactions.

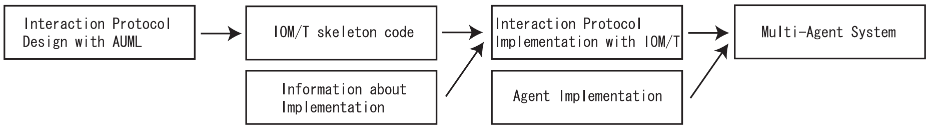

Interaction Protocol Design with AUML
IOM/T skeleton code
Information about Implementation
Interaction Protocol Implementation with IOM/T
Agent Implementation
Multi-Agent System

Fig. 5. Overview of Implementation Process

In the Analysis and Design phase, we acquired the interaction protocols and represented them using AUML sequence diagrams. Here, we show the process of implementation using AUML sequence diagrams. The overview of this process is as follows: First, we use tools to automatically generate skeleton codes of IOM/T from AUML sequence diagrams. Next, we add detail information of the implementation to the code. Then we implement agents and add the mappings of the roles' functionalities. Finally, we use a preprocessor to generate Java code from these codes for the target agent-platform. The overview of the Implementation phase is shown in Fig.5.

AUML to IOM/T. Interaction protocols described in IOM/T have clear correspondence with AUML sequence diagrams. For example Fig.6 shows the correspondence in the Ping protocol case. Thus, we can convert AUML sequence diagrams into skeleton codes of IOM/T based on the following simple algorithms:

1. Each time-line is extracted as a role.
2. A single role action is extracted from a set of zero or more continuous message reception and one or more continuous message transmission within the same time-line.
3. Loop structures are extracted as "while" structures.

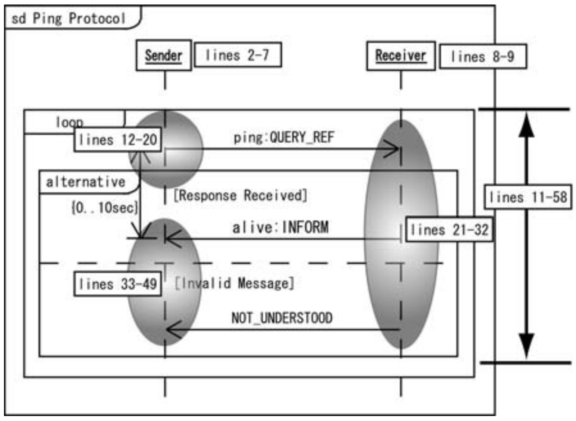


Fig. 6. Correspondence of AUML sequence diagrams to IOM/T

Implementation in IOM/T. The generated skeleton codes of the interaction protocol do not have sufficient information for the implementation. For instance, constraints in AUML sequence diagrams may not contain complete information for execution. Furthermore, AUML sequence diagrams represent the order of messages, but they may not represent the contents of messages. They also lack the information about how the contents are prepared. We have to make a decision on the precise meaning of the constrains and the contents of the messages. So, we add detailed information of the implementation in the code. We refine roles by adding roles' functionalities and information. We refine the flow of the interaction protocol by adding the conditions of "while" structure. Then, we also refine the roles' actions by using Java notation.

In the example of the Ping protocol, we add three functionalities and one variable to the Sender role as shown on lines 3-6 in Fig.2. We add the condition of the "while" structure on line 11. We refine the roles' actions as shown on lines 24-30 and 43-48.

IOM/T to Agent-Platform Specific Java Codes While the interaction protocol is implemented using IOM/T, role functionalities i.e. agent functionalities depend on the specific agent-platform. Codes is merged and converted into target agent-platform Java codes as follows:

1. Generating a FSM model
   A FSM model is generated for each role assuming the role action as a state. State transitions occur on the basis of the order of the role actions and the "while" structures. Furthermore, for roles that do not determine an end of loop, an action is added for that purpose.
2. Implementing a FSM model on the target agent-platform.
   The generated FSM model is converted into the agent-platform specific Java codes.
3. Implementing Role Actions
   Portions implemented in Java are achieved directly. Only special extensions must be converted into agent-platform specific forms. For each role, an interface for accessing the role functionalities is generated. The using of role functionalities is converted into the method invocation through the interface. For each agent, the preprocessor generates helper class which implement interfaces rolls that it play. The helper class delegates role functionalities to the agent class according to the functionality mappings. The extension for FIPA ACL is converted into agent-platform specific representations. The beginning of a protocol is also converted into agent-platform specific representations.

Testing. We can apply unit test to the interaction protocols and the agents. The agent unit test is similar to one for objects. We prepare the test cases for verifying the functionalities of the roles that the agents play. We prepare the test case agents for the unit test for interaction protocols. For each role, we implement test case agents who have the role functionalities. Then, we prepare the test cases that contain the combination of the test case agents and the corresponding result of the interaction. The traditional unit test can check whether each agent is implemented as expected. However, it seems to be hard to check whether the agents can communicate with each other as expected. The unit test for interaction using IOM/T will support this verification. Furthermore, providing the finite ranges of values of the role functionalities, we can generate all possible test case agents. So we can verify the invariants of the interaction automatically.

4 Evaluation

In this section, we evaluate the effectiveness of IOM/T comparing an interaction protocol code described using IOM/T with the same protocol described using JADE[5], a Java based agent-platform. In JADE, interaction protocols are described in the classes which are inherited from the Behaviour class. Until the $done()$ method returns true, the $action()$ method is executed repeatedly. Fig.7 shows the Ping protocol described using JADE.

Firstly, we compare the development where the skeleton code of JADE are directly generated from AUML sequence diagrams to the development where we use IOM/T as an intermediate language. As Huget described in [9], the skeleton code of JADE can be generated directly. We think we have to modify the generated codes by hand in order to

```
class PingSenderBehaviour extends Behaviour {

  private boolean finished = false;
  private int state = 0;

  public PingSenderBehaviour(Agent a) {
    super(a);
  }
  public void action() {
    switch (state) {
    case 0: {
      ACLMessage msg = new ACLMessage();
      msg.setPerformative(ACLMessage.QUERY_REF);
      msg.setReciver(
        ((PingSender)myAgent).getTarget());
      msg.setContent("(ping)");
      myAgent.send(msg);
      state = 1;
      break;
    }
    case 1: {
      ACLMessage msg =
          myAgent.blockingReceive(10 * 1000);
      if (msg == null) {
        ((PingSender)myAgent).knowAsDead();
      } else {
        String content = msg.getContent();
        if (msg.getPerformative()
              != ACLMessage.INFORM ||
          !content.equals("(alive)") {
          ((PingSender)myAgent).knowAsDead();
        }
      }
      state = 2;
      break;
    }
    case 2: {
      if (((PingSender)myAgent).isContinue()) {
        state = 0;
      } else {
        finished = true;
      }
        break;
      }
    }
  }

  public boolean done() {
    return finished;
  }
}

class PingReceiverBehaviour extends Behaviour {

  private boolean finished = false;

  public PingReceiverBehaviour(Agent a) {
    super(a);
  }

  public void action() {
    ACLMessage  msg = myAgent.blockingReceive();
    if(msg != null){
      ACLMessage reply = msg.createReply();
      if(msg.getPerformative()
           == ACLMessage.QUERY_REF){
        String content = msg.getContent();
        if (content != null &&
            content.equals("(ping)")){
          reply.setPerformative(
                             ACLMessage.INFORM);
          reply.setContent("(alive)");
        } else {
          reply.setPerformative(
            ACLMessage.NOT_UNDERSTOOD);
          reply.setContent(msg.toString());
        }
      } else {
        reply.setPerformative(
                   ACLMessage.NOT_UNDERSTOOD);
        reply.setContent(msg.toString());
      }
      myAgent.send(reply);
    }
  }

  public boolean done() {
    return finished;
  }
}
```

Fig. 7. Iterated Ping protocol represented in JADE

represent the information which is showed as notes or constraints in AUML sequence diagrams in either case, although we tend to cause a mistake in this operation. For example, in Fig.1, there is a constrain, "[Valid Response Received]". In JADE code, this constraint is represented on lines 27-30 in Fig. 7, where we check the content of ACL message. The creation of this message is represented on lines 70-72. Although these two parts have strong relationship, they completely exist in separate code. On the other hand, in IOM/T, the constraint is represented on lines 37-39 in Fig. 2, and the creation of message is represented on lines 25-26. In this case, these two parts exist in a single unit of code. Furthermore, the role actions that include these parts are closed by. Therefore, it seems to be better to modify the skeleton codes of IOM/T rather than the skeleton codes of JADE.

Next, we consider the *understandability* . If we compare the IOM/T code with the JADE code, there are two significant differences. One difference is the code scattering for an interaction protocol. If we use JADE, we describe a code of the interaction protocol for each role. The code contains the only own part. Then, the codes for interaction protocol dispersed. If we use IOM/T to describe, the interaction protocol is described in a single unit of code. As a result, the correspondence between the message transmission and the message reception is clear. In Fig.7, the correspondence of line 17 to line 64 can easily be confused. However, In Fig.2, the same correspondence is represented on line 17 and 22. It is easy to understand this correspondence. The other difference is the representation of the interaction protocol flow. In the JADE code, we have to realize state transitions by assuming integers as states since Java has no notation for representing state transition. We can describe complex state transitions with this method. However it is not easy to understand the interaction protocol flow from the code, since we have to understand each state and each transition. Especially, loops are not clear since they are represented as recursive structures as shown on line 39. On the other hand, in the IOM/T code, the flow of the interaction protocol is represented in natural manner. Each role action corresponds to a state. The state transitions are represented on the basis of the order of the occurrences of the role actions, and the loops are represented explicitly using "while" structures. So we can intuitively understand the flow. Indeed we can represent more complex state transitions with the former method, but we mostly do not need to describe very complex state transitions. In this paper, we do not show the details of Analysis and Design phase. We think we can get the designs of interactions that are relatively simple through the phase. We can build up the system by using sub-interactions. So, our current achievement is to represent interaction protocols based on relatively simple AUML sequence diagrams.

IOM/T also improves the *maintainability* . IOM/T enhances the understandability of the interaction protocol as mentioned above. So, when we have to commit some changes, we can easily understand where and how we have to change. For example, suppose there was a change such as the "$alive$" message is replaced with the "$pong$" message in the Ping protocol. In Fig.7 the changes will be applied to line 30 in the $PingSenderBehaviour$ class and to line 72 in the $PingReceiverBehaviour$ class. These changes are not obvious. On the other hand, the same modification is easily achieved in the IOM/T code. We will change the lines 25 and 39 in Fig.2. Since the role actions that contain these lines are close by, the correspondence of the code is obvious.

In other words, since our development process holds *traceability* , the maintenabilty is improved. An AUML sequence diagram corresponds to a single unit of IOM/T code. A message in the diagram corresponds to role actions which are closed by. Then, we can found out the corresponding part of the code from the changes in the design model. So, we can deal with the changes easily.

5 Related Work

So far, several interaction protocol description languages such as AgenTalk[12], Q[11], COOL[2], COSY[4], micro-protocol[15] have been proposed. Each of these languages has specific interesting features. Micro-protocol is a language designed to improve reusability. This language is not considered here since we mainly consider languages for implementation. AgenTalk, COOL, COSY and Q are languages for implementation. However, with these languages, an interaction protocol is dispersed and represented within the scenario of each agent. Our aim is to bridge the gap between the AUML design and the implementation, by treating the interaction protocol as an independent module in the implementation.

In the area of interaction protocols, several results have been achieved on the basis of the formal methods. Huget have provided the methods for verifying AUML sequence diagrams by using Spin[10]. Paurobally have proposed diagrams as counterpart of formal methods in order to bridge the gap between formal specification and intuitive development[13]. These endeavors have relationship with our research. However, in this paper, we proposed a novel language and implementation process.

There are several methods for generating skeleton code from graphical representations. Huget shows some elements for generating code from AUML sequence diagrams[9]. Tahara provides the IPEditor[14] which generates skeleton codes of agents from AUML based graphical notation. Dinkloh has developed tools for generating skeleton codes of Jade from AUML sequence diagrams[6]. These works provide methods for generating object oriented programming code from design. Our approach deals with interaction protocols as it is in the Implementation phase.

6 Conclusion

In this paper, we proposed a new interaction protocol description language, IOM/T. We also showed how the implementations are conducted based on the AUML sequence diagrams. By applying this approach, we will be able to bridge the gap between AUML and implementation. Moreover, the representation in IOM/T allows easy understanding of interaction protocol flows and improves maintainability.

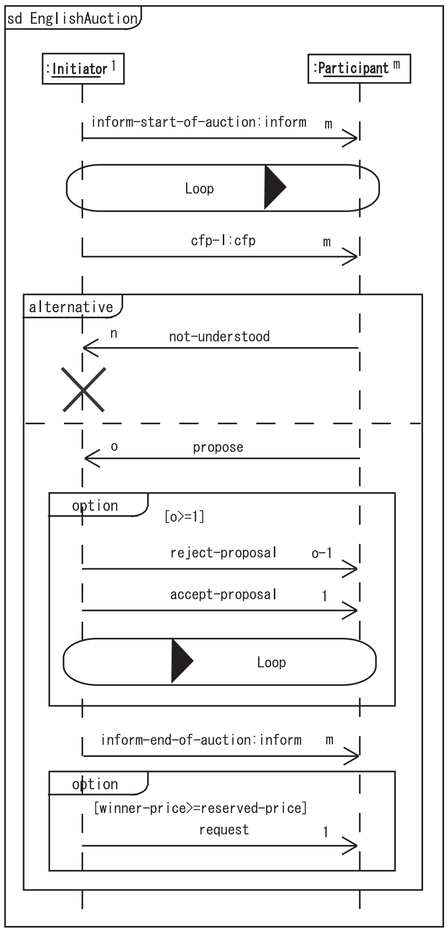
This work is currently in progress. We are developing an Integrated Development Environment for this approach, and verifying the validity of this approach through development examples. We will also investigate the inclination of IOM/T thruogh the development examples. In addition we are considering the needs for a formal semantics of IOM/T. Finally we plan to extend IOM/T to make IOM/T equivalent to AUML sequence diagrams.

References

1. Amal El Fallah-Seghrouchni Alexandru Suna. A mobile agents platform; architecture, mobility and security elements. In *Programming Multiagent Systems languages, frameworks, techniques and tools ProMAS 2004 Workshop*, 2004.
2. M. Barbuceanu and M. S. Fox. Cool: A language for describing coordination in multiagent systems. In Victor Lesser and Les Gasser, editors, *Proceedings of the First International Conference oil Multi-Agent Systems (ICMAS-95)*, pages 17–24, San Francisco, CA, USA, 1995. AAAI Press.
3. Lars Braubach, Alexander Pokahr, Winfried Lamersdorf, and Daniel Moldt. Goal representation for bdi agent systems. In *Programming Multiagent Systems languages, frameworks, techniques and tools ProMAS 2004 Workshop*, 2004.
4. B. Burmeister, A. Haddadi, and K. Sundermeyer. Generic, configurable, cooperation protocols for multi-agent systems. *Lecture Notes in Computer Science*, 957:157–, 1995.
5. CSELT. JADE. `http://sharon.cselt.it/projects/jade/`.
6. M. Dinkloh and J. Nimis. A tool for integrated design and implementation of conversations in multi-agent systems. In *Programming Multiagent Systems languages, frameworks, techniques and tools ProMAS 2003 Workshop*, 2003.
7. FIPA. Agent UML. `http://www.auml.org/`.
8. FIPA. FIPA Modeling: Interaction Diagrams. `http://www.auml.org/auml/documents/ID-03-07-02.pdf`.
9. Marc-Philippe Huget. Generating code for Agent UML sequence diagrams. Technical report, University of Liverpool Department of Computer Science, 2002.
10. Marc-Philippe Huget. Model checking Agent UML protocol diagrams. Technical report, the department of computer science, University of Liverpool., 2002.
11. Toru Ishida. Q: A scenario description language for interactive agents. *IEEE Computer*, 2002.
12. K. Kuwabara, T. Ishida, and N. Osato. Agentalk: Describing multiagent coordination protocols with inheritance. 1995.
13. Nicholas R. Jennings Shamimabi Paurobally. Developing agent interaction protocols using graphical and logical methodologies. In *Programming Multiagent Systems languages, frameworks, techniques and tools ProMAS 2003 Workshop*, 2003.
14. Y. Tahara, A. Ohsuga, and S. Honiden. Mobile agent security with the IPEditor development tool and the Mobile UNITY language. In *Proc. of Agents 2001*, pages 656–662. ACM Press, 2001.
15. B. Vitteau and M.-P. Huget. Modularity in interaction protocols. In *Agent communication languages and conversation policies AAMAS 2003 Workshop*, 2003.
16. Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

A English Auction

A.1 English Auction Protocol Represented in AUML Sequence Diagram



A.2 EnglishAuction Protocol Described in IOM/T

```
protocol EnglishAuction {

  player Initiator {
    List getParticipants();
    boolean isEndAuction();
    String getCurrentPrice();
    ACLMessage selectBids(List bids);
    AID getWinner();
    List participants;
    Date cfpTime;
  }
```

```
player * Participant {
  void knowBeginAuction();
  boolean isBid(String item, String current);
  boolean
    isUnderstandable(String item, String current);
  String getBidPrice(String item, String current);
  void knowPrice(String price);
  void knowAccept();
  void knowReject();
  void knowRequest(String req);
  boolean isBid;
  boolean isAccept;
}

interaction {
  player Initiator {
    participants = getParticipants();
    ACLMessage msg = new ACLMessage();
    msg.setPerformative("INFORM");
    msg.setContent("(inform-start-of-auction)");
    msg.setReceiver(participants);
    sendAsync(msg);
  }

  player Participant {
    ACLMessage msg = recvBlock();
    knowBeginAuction();
  }

  while (!Initiator.isEndAuction()) {

    player Initiator {
      ACLMessage msg = new ACLMessage();
      msg.setPerformative("CFP");
      msg.setContent(getItem() + ":" + getCurrentPrice());
      msg.setReceiver(participants);
      sendAsync(msg);
      cfpTime = Calender.getInstance().getTime();
    }

    player Participant {
      ACLMessage cfp = recvBlock();
      String[] str = msg.getContent().split(":", 2);
      String item = str[0];
      String current = str[1];
      isAccept = false;
      if (isUnderstandable(item, current)) {
        ACLMessage res = cfp.createResponse();
        res.setPerformative("NOT_UNDERSTOOD");
        res.setContent(cfp.getContent());
        sendAsync(res);
        terminateProtocol();
      }
      isBid = isBid(item, current);
      if (isBid) {
        String price = getBidPrice(item, current);
        ACLMessage res = cfp.createResponse();
        res.setPerformative("PROPOSE");
        res.setContent(price);
        sendAsync(res);
      }
    }

    player Initiator {
      List bids = new LinkedList();
      Calender cal = Calender.getInstance();
      while (cal.getTime().getTime() - cfpTime.getTime() < 10 * 1000) {
          ACLMessage bid = recvNonBlock();
          if (bid.getPerformative().equals("NOT_UNDERSTOOD")) {
            participants.remove(bid.getSender());
          } else if (bid.getPerfromative().equals("PROPOSE")) {
            bids.add(bid);
          }
        }
        ACLMessage accept = selectBids(bids);
        if (accept != null) {
          Iterator itr = bids.iterator();
          while (itr.hasNext()) {
            ACLMessage bid = (ACLMessage)itr.next();
            ACLMessage msg = bid.createResponse();
            msg.setContent(bid.getContent());
            if (accept == bid) {
              msg.setPerformative("ACCEPT_PROPOSAL");
              sendAsync(msg);
            } else {
              msg.setPerformative("REJECT_PROPOSAL");
              sendAsync(msg);
            }
          }
        }
      }

      player Participant {
        if (isBid) {
          ACLMessage msg = recvBlock();
          if (msg.getPerformative().equals("ACCEPT_PROPOSAL")) {
            knowAccept();
            isAccept = true;
          } else {
            knowReject();
          }
        }
      }

    }

    player Initiator {
      ACLMessage msg = new ACLMessage();
      msg.setPerformative("INFORM");
      msg.setReceiver(participants);
      msg.setContent(getCurrentPrice());
      sendAsync(msg);
      ACLMessage msg = new ACLMessage();
      msg.setPerformative("REQUEST");
      msg.setReceiver(getWinner);
      msg.setContent(getCurrentPrice());
      sendAsync(msg);
    }

    player Participants {
      ACLMessage msg = recvBlock();
      String price = msg.getContent();
      knowPrice(price);
      if (isAccept) {
        ACLMessage req = recvBlock();
        knowRequest(req.getContent);
      }
    }

  }
}
```

Inter-agent Communication in IMAGO Prolog

Xining Li and Guillaume Autran

Department of Computing and Information Science,
University of Guelph,
Guelph, Canada
xli@cis.uoguelph.ca

Abstract. A mobile agent application often involves a collection of agents working together for a common task. For cooperation among agents to succeed, an effective inter-agent communication framework is required. This paper describes the design of the communication mechanism in IMAGO Prolog. IMAGO Prolog is a variant of Prolog with an extended API for intelligent mobile agent applications. It deploys mobile messengers for inter-agent communication. Messengers are anonymous, thin agents dedicated to deliver messages. A messenger can move, clone, and make decisions for its assigned task: track down the receiving agent and reliably deliver messages in a dynamic, changing world. Moreover, agent communication language is purely declarative and consistent with the syntax, semantics and pragmatics of Prolog. As a result, mobile agents exchange information and achieve synchronization through first order logic terms and unification.

1 Introduction

A mobile agent is a self-contained computing entity, roaming the internet to access data and services, and carrying out its assigned decision-making and problem-solving tasks on behalf of a user. A mobile agent application usually consists of a collection of agents working together for a common task. Agents are not working alone in most applications, instead, they need to communicate with each other for exchanging information and achieving synchronization. For cooperation between agents to succeed, an effective inter-agent communication framework is required, which includes the design of a communication model, an inter-agent communication language and protocol.

Generally speaking, communication models are concerned with conceptual paradigms such as RPC/RMI, message-based, or event-based, whereas communication language and protocol deal with problems such as how to represent information and knowledge, how to name mobile agents, how to establish communication relationships, how to track moving agents, and how to guarantee reliable message delivery.

We present in this paper a promising agent-based communication model explored in the design and implementation of IMAGO Prolog. The idea is to deploy intelligent mobile messengers for inter-agent communication. Messengers are thin agents dedicated to deliver messages. Like normal agents, a messenger can move, clone, and make decisions. Unlike normal agents, a messenger is anonymous and its special task is to track down the receiving agent and reliably deliver messages in a dynamic, changing environment. In addition, agent communication language is purely based on the first order logic, *i.e.*, consistent with the syntax, semantics and pragmatics of Prolog. Communicating mobile agents simply use logic terms and unification to exchange information and achieve synchronization.

This paper is organized as follows. Section 2 gives a briefly review of recent works related to this research, focusing on the agent communication models and agent communication languages. Section 3 presents the intelligent mobile messenger model as well as feasible solutions for problems such as agent naming, agent tracking, and communication predicates. In section 4, we show examples of inter-agent communication patterns. IMAGO Prolog is currently being implemented as an experimental prototype system. An evaluation release of the IMAGO system is available at the IMAGO Web site [1]. Finally, we present the conclusion and outline of future work.

2 Related Work

Processes in a distributed system must interact with each other using some kind of communication models to exchange data and coordinate their execution. Several communication models have been widely used in distributed systems as well as most mobile agent systems. Typical models are *Message Passing*, *Remote Procedure Call (RPC/RMI)*, and *Distributed Event Handling.*

Message passing is used to support peer-to-peer communication patterns and is the most adopted model in mobile agent systems such as Aglet[1], Mole[2], D'Agent[3], Voyager[4], SyMPA[5],*etc.* Aglet supports an object-based messaging framework that is flexible, extensible, rich, and both synchronous and asynchronous. Mole deploys the (global) exchange of messages through a session-oriented mechanism. Agents that want to communicate with each other must establish a session before the actual communication can start. D'Agent supports text-based message passing. The location and identity of the receiver should be known by the sender. There is no guarantee for reliable message delivery because communication is lost as soon as one peer jumps to another location. Voyager implements message passing through the concept of virtual objects. Agents are a special type of object in a Voyager application. Communication with a remote object is handled by its virtual object which hides the remote location and acts as a reference of the remote object. When messages are being sent to the remote agent, the virtual object forwards the message to the remote object and returns messages back if necessary. SyMPA is a mobile agents platform supporting intelligent agents developed in CLAIM[6], a declarative agent-oriented programming language. In this platform, each agent has a queue for storing incoming

[1] http://draco.cis.uoguelph.ca/main.html

messages. Message passing among agents can be unicast, multicast or broadcast asynchronously.

RPC and RMI are commonly used paradigms in today's distributed programming. Since there is no distinction in syntax between an RPC and a local procedure call, it provides access transparency to remote operations. Several mobile agent systems support RPC/RMI paradigm such as Mole and Voyager. An argument against this paradigm is that under the new paradigm where mobile agents can move to any remote host for data and services, why we need RPC/RMI at all. Agents for Remote Action (ARA)[7] attempts to minimize the remote communication through a meeting oriented paradigm. ARA provides client/server style interaction between agents. The core provides the concept of a service point which is the meeting point with a well known name where agents located at a specific place can interact as clients and servers through an RPC-like invocation on a local host.

The concept of event based communication and synchronization can be viewed as a sophisticated paradigm of meeting oriented agent coordination. Some mobile agent systems have much in common with those event frameworks employed in GUI toolkits supported by Java and Tcl/Tk. Mobile agent systems such as Concordia[8], D'Agent, Mole, *etc.*, extend the event-driven programming technique to coordinate groups of mobile agents. In this paradigm, agent synchronization is achieved by the objects that are defined as active entities responsible for the coordination of an entire application or parts of it. These synchronization objects could be user defined objects or system implemented event managers. It is their responsibility to accept event registration, listen and receive events, and notify interested parties when an event arises. On the other hand, an agent participating in such groups is responsible to register a list of event types it is interested in as well as the location it wishes events to be sent. Certainly, this model requires the static binding of agents with their registered locations, or otherwise event notification becomes unreliable.

Nevertheless, no matter which inter-agent communication model is selected, the model must be implemented through a stack of dedicated communication protocols. Interprocess communication is dependent on the ability to locate the communication entities. This is the role of the naming services which primarily map each entity in its name space to a fixed location in traditional distributed systems. Mobile agents are distributed processes. However, once they are invoked they will autonomously decide the hosts they will visit and the tasks they have to perform. Their behavior is either defined explicitly through the agent code or alternatively defined by an itinerary which is usually modifiable at runtime. As a result, the mobility of agents makes it much harder to provide such kind of name resolution service because there is virtually no way to bind a moving agent with a static (fixed) location. Thus, existing MA systems either do not provide the ability of automatically tracking moving agents, or overly constraint the mobility of agents. For example, Aglet API does not support agent tracking. Instead, it leaves this problem to applications. To avoid tracking agents during communication, Mole prevents agents from moving if they are involved in a session.

Briefly speaking, locating an agent is invoking a function of "where_is(A)" which should return the current address (or access point) of A. Researchers have recently proposed many schemes for designing such a function. Various approaches for storing, updating, and locating mobile agents are well addressed in [9][10].

Some communication protocols use *broadcasting* or *multicasting* approaches to locate mobile agents. In this paradigm, location queries are broadcast to the entire network, or multicast to a specific group of hosts. Upon receiving such a request, each machine must check the names of hosting agents and give a response if the requested name is found. This paradigm is mainly applicable to intranets. It becomes inefficient in a large-scale network. Furthermore, the answer to "where_is(A)" is not accurate if A moves to another host immediately after the answer is returned, which makes message loss unavoidable. To solve this problem, a snapshot broadcasting strategy [11] was proposed. However, this scheme requires agent communication and migration through a strict FIFO channel, inhibits the movement of the located agent during message passing, and involves a considerable system overhead.

Another popular approach is to use a fixed location server, called *home*, to keep track of locations of mobile agents. In this scheme, agents follow a triangular routing to communicate with each other, that is, a message is sent to *home* first, which looks up the destination address and then simply forward the message to the receiving agent. Unfortunately, the same problem remains. The address returned from the lookup function "where_is(A)" is ambiguous: A might still reside at that address, or A might have moved to another host and its location updating packet is on the way to home, or A might even have started to move at the same time a message is sent to its current location.

Forward pointers is a promising alternative for locating mobile agents. This scheme does not depend on a "where_is(A)" lookup function. Instead, each mobile agent host keeps a reference (forward pointer) for each moving agent. For example, in Voyager, a virtual object keeps track of the remote object by its last known address. If the remote object moves from its last location, it will leave a *secretary* object behind to forward messages to its new location. The secretary object will be removed only if a returned message has been received by the corresponding virtual object. The advantage of this approach is that it could automatically track down moving agents. However, it could cause a lot of overhead and delay if remote objects involve frequent movements. Furthermore, a theoretical flaw is that messages might forever chase a frequently moving receiver, even though this hardly occurs in practice.

Mailbox-based scheme [12] is a set of mobile agent tracking protocols proposed recently. In this approach, each mobile agent is assigned with a mailbox to buffer incoming messages. A mailbox is a mobile object and could be detached from its owner, but it is not an agent-like object in the sense that it can not make any decisions. inter-agent communication follows the triangular manner such that the mailbox serves as the middle access point between sender and receiver. This scheme offers a 3D design space which includes *home* protocol, *forward pointer* protocol and *distributed registration* protocol. It also allows users to select a parameterized protocol ranging from unreliable communication to reliable communication. However, protocols which guarantee reliable message delivery require expensive synchronization and location registration.

Communication between agents takes place by means of an *Agent Communication Language (ACL)*. The essence of an ACL is to make agents understanding the purpose and meaning of their messages. In an ACL, a message consists of two separate aspects, namely, *performative* and *content*. The performative shows the purpose of a message while the content gives a concrete description for achieving the purpose. Of course, the sending agent and the receiving agent must agree not only on the ACL but also on the content language, so that they have at least the same understanding of the purpose and the same interpretation of the content of a message.

Typical research in this area are KQML [13] and FIPA ACL[14]. KQML is both a message format and message handling protocol to support run-time knowledge sharing among agents, it offers a standard ACL that intelligent agents can use to communicate among themselves as well as with other information servers and clients. The syntax of KQML is based on Common Lisp. As a result, a message is represented as a balanced parenthesis list consisting of performative, content, and a set of optional arguments. For example, a message representing a query in standard Prolog about the set of all answers of IBM stock is encoded as:

```
(ask-all
    :content "price(ibm, [Price, Time])"
    :receiver stock-server
    :language standard_Prolog
    :ontology nyse-ticks)
```

FIPA is a forum of 70+ international telecommunication companies and research institutes which specifies open standards focusing on languages and protocols for communication, coordination, and management of heterogeneous agents. In other words, FIPA is more like an abstract architecture which makes all FIPA-compliant systems to communicate and interact with each other through a common ACL.

An ACL developed by FIPA has defined several performatives of messages, such as INFORM, QUERY-IF, CFP, PROPOSE, and so on. For example, performative INFORM indicates that the content of a message is a *true* proposition, whereas QUERY-IF asks if the proposition given as the content of a message is *true*. On the other hand, FIPA does not prescribe the language used to express the message content. Instead, it specifies the ACL Protocol Data Unit(PDU) as a data structure which contains a set of one or more message elements, such as performative, sender, receiver, language, content, *etc.* Precisely which elements are needed for an ACL message is application dependent, except that the performative element is mandatory in all ACL messages. Certainly, most ACL messages will also include sender, receiver, and content elements.

A FIPA ACL-like PDU is given in Table 1.

Table 1. An example of a FIPA ACL-like PDU

Element	**Value**
Performative	INFORM
Sender	*agent@some.server*
Receiver	*database@some.db.server*
Language	Prolog
Ontology	genealogy
Content	*female(mary), mother(mary, tom)*

Although the RPC/RMI model offers access transparency, it turns out that general purpose, high level message-based models are more convenient, and often adopted by most mobile agent systems. However, it can always be argued whether agent communication should be remote or restricted to local, considering that the most attractive motivation of mobile agents is that they are able to migrate between locations to locate data and services as well as their peers, and therefore avoiding remote communication. Furthermore, a communication model is usually implemented by a stack of communication protocols. It is also questionable whether such a protocol is actually necessary if we already have a simple, reliable agent migration protocol. In other words, can we utilize the mobility of agents to achieve powerful, reliable and flexible inter-agent communication?

In the area of inter-agent communication, research looked at languages that are suitable for mobile agent programming and languages for agent communication. The goal of separating agent communication language from agent programming language is to allow agents developed in different programming languages to share information and knowledge through a common ACL. However, it can also be argued how *common* such an ACL could be. For example, both KQML and FIPA ACL are based on the core concept of *performatives*, which determines the kinds of interactions one can have with an agent. Unfortunately, the performative alone can only identify the protocol to be used to deliver the message and to indicate purpose of communication. Another language (usually a specific programming language) must be used to specify the content which gives the concrete description for achieving the purpose. It is clear that an agent in Prolog does not have the same interpretation of the content of a message as an agent in Java, even though they may have the same understanding of the performative. As a result, a common ACL works fine as long as agents in communication knowing how to interpret the symbols in a message. This leads to another question, can we provide a high level programming abstraction which makes the ACL and the agent programming language consistent with each other, or more precisely, sharing a common syntax, semantics and pragmatics?

In response to these questions, the following section will present the agent-based communication model in IMAGO Prolog - Intelligent Mobile Messengers which allow communicating agents to exchange messages through Prolog terms and unification.

3 Naming and Locating Mobile Agents

The inter-agent communication model in IMAGO Prolog has focused on simplicity and ease of use, and at the same time offering more expressive power. To better understand how our model works, we shall first distinguish messengers from normal mobile agents (and we shall call them as *workers* in the rest of this paper). A worker is a normal mobile agent created by its owner for some specific task, whereas a messenger is an anonymous thin agent dispatched by a worker to deliver messages. Generally speaking, a worker is purposely separated from the location of its owner, and best equipped with as much intelligence as possible in order to autonomously carry out the assigned task on behalf of its owner. Unfortunately, adding more intelligence to a worker will make some sacrifices of mobility. Thus, to deploy these thick workers directly for inter-agent communication is neither economical nor practical. This is the reason for introducing messengers - specialized thin agents - which not only provide an agent-based solution for inter-agent communication but also make the solution efficient and feasible.

Clearly, each worker must have a unique name so that its owner and other workers can communicate with it over the network. Some systems, such as Voyager, adopt location-transparent names at the application level. In contrast, systems such as Aglets and D'Agent, assign location-dependent names. For example, an agent in Aglets is associated with a unique identifier so that every agent in the network can be uniquely addressed by combining its identifier with its context URL.

Obviously, identifying an agent by the combination of its identifier and its *current location* does not fit well to the mobility of agents. Since a worker may move any time to an arbitrary remote server, its *current location* is uncertain. For this reason, our model adopts a location-transparent, closed-world naming scheme to identify workers. First, we assume that a user-friendly, symbolic name is assigned to each worker. Such a name must be unique in the application (a closed world) where the worker is created, and immutable throughout the worker's lifetime. This user-friendly symbolic name is used to unambiguously refer to the worker inside the application which it belongs to.

Secondly, we assume that each application is bound to a *home* location which always exists during the lifecycle of the application. Consequently, workers of an application are all originated from the same home. If several applications are concurrently running at the same home, we shall use different sequencing numbers to distinguish them. Therefore, by concatenating the user-friendly symbolic name, the home URL and the application sequencing number, we have a location-transparent, globally unique identifier for each worker. It is worthwhile to note that the home URL embedded in a worker's identifier is independent from the worker's current location. By using such identifiers, it is sufficient to unambiguously refer to a worker over the internet.

Our next assumption is that a stack of protocols exists to support efficient, reliable and strong agent migration. There is no constraint on the freedom of agent mobility. Both workers and messengers are allowed to move freely from one host to another. That is, they may decide where to go based on their own will or the information they have gathered.

In IMAGO Prolog, deployment of messengers is the only way to achieve inter-agent communication. Since messengers are in fact mobile agents, they can be designed to serve different purposes such as asynchronous messaging, synchronous messaging, broadcasting or multicasting, *etc.* To accommodate incoming messengers, we assume that each worker is associated with a messenger queue which holds all messengers destined to this worker and waiting for delivery of messages. The assumptions we made in this section are reasonable because they are already satisfied by our implementation as well as some other mobile agent systems.

In order to locate a moving worker, agent servers should maintain enough information to keep track of current location of every worker. However, we have indicated that it is virtually impossible to have the precise information about a changing environment, because an application may involve workers which are creating, cloning or moving all the time. To cope with such a dynamic configuration, our model maintains heuristic location information through *distributed registration* and *local updating* operations, and employs a variant of *forward-pointer-based* approach plus a *home-based* mechanism as the backup.

As we explained before, identifiers of workers have an embedded static *home* location, although these workers might spread and roam over the network. This home is the default server for workers to send their registration. A new born worker, either by creating or cloning and regardless of born at the home host or a remote host, must register its birth place with the home automatically. Even though registrations take a distributed manner, *i.e.*, registration messages might flow to the home from different remote hosts, it does not cause much network traffic because each worker registers only once in its whole lifecycle.

A registration message is stored as a *worker record* in the local cache of the home server. A worker record is a structure of the form

$$\{worker\_id, timestamp, status\}$$

where *worker_id* is the globally unique identifier of the worker, *timestamp* gives the time the record last been modified, and *status* indicates the current state of the worker. For the sake of simplicity, we assume that a worker must be in one of three possible states: ALIVE, DISPOSED, or MOVED_TO(url).

Like the home server, a remote agent server also remembers a collection of worker records per application basis. However, it maintains caching information through the *local updating* operation. Such an operation is very efficient because it is done completely in the local system layer. In general, a worker record is inserted into the local cache when the worker is created or cloned locally, or the first time it moves into this server. To make the local caching more effective in locating a worker, a remote server should also cache sender's information carried by a messenger. Obviously, caching sender's information exploits locality. For instance, a receiving worker is most likely to reply to its sender in the near future, and the sender's location can be found immediately from the local cache.

An *updating* operation is also applied to a worker record whenever the worker changes its state. For example, when a worker moves from host X to host Y, its cached record at X is modified with the new state MOVED_TO(Y). This is very similar to the *forward pointer* scheme which leaves behind a forwarding reference whenever an entity moves to a new location. Likewise, if the worker moves from Y back to X, its record at X will be simply changed back to the state ALIVE. Furthermore, updating operations can be used to short-cut a forwarding chain. For example, suppose that a worker moves from X to Y and then to Z. From there, the worker dispatches a messenger to a receiver at X. When the messenger arrives, the updating operation will change the worker(sender)'s record from the old state MOVED_TO(Y) to the new state MOVED_TO(Z). Therefore, subsequent communications to that worker will be dispatched to Z directly.

In our model, we do not intend to have a network-wise "where_is(A)" lookup function for locating the *current* address of A. Instead, we use a local lookup function which returns a *possible* location of A. The reason for saying *possible* is that the information recorded in a server's local cache is heuristic. For instance, if the status of a worker is recorded as MOVED_TO(Y), there is no guarantee that the worker we are looking for is still working at Y, because a worker is never bound to an absolute host address - it may very well have moved on to another location. However, it is guaranteed that successive lookup's at subsequently forwarded heuristic hosts will eventually trap the worker if the worker really wants to accept the messenger.

Now, let us consider the general lookup facility for remote servers. The principle is very simple. We only search the local cache to find where the worker *possibly* resides in. This lookup function will never return something like WORKER NOT FOUND. Instead, it either returns the value of the current *status* from the located worker record, or a special *status* MOVED_TO(*home*) if a cache miss occurs. Since a remote server might host multiple concurrent agents (workers and messengers), the lookup operation and the updating operation must be mutual exclusive. That is, when a messenger has to locate a worker and deliver message, it must lock the cached worker record (critical region) to achieve mutual exclusion and ensure that the worker is not able to change its state at the same time.

The lookup function on the *home* server is analogous to the above description. In principle, there is no cache miss because the home should hold a complete set of worker records. However, what possibly happens in practice is that a messenger is dispatched to a worker who might have not been created yet or whose registration message might be on the way home. To solve this problem, the lookup function simply blocks this messenger. A blocked messenger will be resumed if a new registration with a matching receiver arrives.

A messenger is an agent. It has its own code to be executed. There are many ways to design messengers for different purposes. To make it easier to understand, we will start out by discussing a very important system primitive. Then we will look at a simple messenger and discuss its behavior in some detail.

System primitives serve as the interface between agents and the underlying system. In addition to the commonly used primitives such as *create*, *move*, *clone*, *etc.*, another primitive which plays a major role in between workers and messengers is *attach*. The following code segment shows the skeleton of the *attach* primitive.

```
attach(receiver){
        lock(local_cache);
        r = lookup(receiver);
        if (r == ALIVE){
            // insert this messenger into
            // the receiver's messenger queue
            unlock(local_cache);
            // context switch to another ready agent
        }
        else {
            unlock(local_cache);
            return r;
        }
}
```

The basic idea behind a messenger is try to track down the receiver until its message is accepted. To achieve such behavior, a messenger simply invokes the following recursively defined *deliver* function.

```
deliver(receiver, message){
        r = attach(receiver);
        if (r == RECEIVED || r == DISPOSED)
            dispose();
        else { // r == MOVED_TO(url)
            move_to(url);
            deliver(receiver, message);
        }
}
```

A messenger starts by invoking a call to *attach* which will issue a lookup mutual exclusively. Only two possible cases make the *attach* return immediately: the receiver has deceased locally, or the receiver has moved to another host. Recall that MOVED_TO(*home*) will be returned if a cache miss occurs in a lookup, so that it seems as if the receiver has migrated to home. Therefore, the messenger will either follow the receiver by calling *move* and then try to deliver at the new host, or simply dispose itself if the receiver no longer exists.

On the other hand, if the receiver is ALIVE at the current host, the *attach* primitive will insert the caller into the receiver's messenger queue and then make it *suspended*. The underlying system is free to schedule another runnable agent to execute. It is now the receiver's responsibility to resume an attached messenger based on different actions it is going to perform. Certainly, the receiver might invoke an *accept*-like primitive. If this happens, the accepted messenger is resumed as soon as its carried message has been transferred to the worker's working space. Unfortunately, whether attached messengers will be accepted by its receiver is unknown, because the receiving worker might not be ready to accept any messenger yet. For instance, it is possible that the worker moves to another host while there are pending messengers. It is possible that the worker disposes itself without accepting messengers. It is also possible that the worker clones itself while it has a non-empty messenger queue. Nevertheless, these possible cases are facing with the same problem, namely, how the receiver deals with pending messengers.

From the well-known semantics of *strong migration*, a mobile agent should take its code, data and execution state together when it moves to a new location. Unfortunately, such semantics have a flaw of ignoring messages. In fact, messages to an agent should also be a part of the agent. If they have been received, they become a portion of data. If they have not been received (either buffered by the underlying system or still in transmission), then they should go with the agent together whenever the agent moves. Therefore, the highest degree of *strong migration* is to take four parts of an agent, *i.e.*, code, data, state and messages, into consideration.

Although it sounds more difficult, the solution in our model is straightforward. A worker simply resumes all attached messengers if it moves. Likewise, a worker resumes all pending messengers if it disposes itself. Now consider what happens, for example, when a messenger is resumed by the receiver it was attached to. At this point, it seems as if the call to *attach* has just returned. However, the returned value might be one of the three possible cases now: RECEIVED, DISPOSED or MOVED_TO(url). Therefore, the resumed messenger must be able to cope with different cases and try to re-deliver the message if the message has not been received yet and the receiver is still alive.

This is why a messenger will invoke *attach* each time it moves to a new place. A messenger claims that "I can track the receiver down provided I have the trail of the receiver", whereas our lookup facility says that "the location I found is where most likely the receiver resides at, or at least the receiver has lived". In other words, the heuristic location from the lookup facility provides the trail of the receiving worker while leaves the tracking-down job to the messenger.

4 Inter-agent Communication

To be consistent with the standard Prolog, the IMAGO Prolog introduces system directives for agent module declarations. For example, the following code segment specifies that the Prolog text bracketed between the pair of directives *worker(...)* and *end_worker(...)* defines an agent prototype named as *stock_buyer*.

```
:- worker(stock_buyer).
        stock_buyer(Arg) :-
            worker_body, ...
:- end_worker(stock_buyer).
```

Like other logic programming systems, IMAGO Prolog Application Programming Interface(API) is presented as a set of builtin predicates. This set consists of builtin predicates common to most Prolog-based systems and new builtin predicates specially designed for mobile agent applications [15]. Typical predicates include *create* which spawns an instance of mobile agent from a given worker prototype, *clone* which forks a duplicate of the calling worker, and *move* which allows an agent to migrate to a destined host.

At any stage of execution, a worker can *dispatch* a messenger to deliver a message to another worker. Predicate *dispatch* takes three arguments: the name of a messenger prototype, the name of receiver, and the message represented by a Prolog term. A messenger has its own code to be executed, which includes tracing and locating the receiving worker, and delivering message.

There are many ways to design messengers for different purposes. To make messengers easy to use, the IMAGO Prolog provides a set of system messengers as a part of its API. For example, the current release of IMAGO Prolog (version 1.0) supports the following list of system messengers.

$oneway_messenger: delivers a message to the receiver;
$multicasting_messenger: generates multiple clones to deliver a message to a group of receivers in parallel;
$postman_messenger: sequentially delivers a list of mails to a list of workers;
$paperboy_messenger: sequentially delivers a copy of message (like a copy of newspaper) to a list of subscribers;
$cod_messenger: delivers a message to the receiver and carries the message unification back to the sender (Cash On Delivery);
$collecting_messenger: sequentially collects unification of a list of variables from a list of workers and carries the unification results back to the sender.

The main advantages of system messengers lie in their reliability and efficiency. System messengers are carefully designed and fully tested. The varieties of system messengers are sufficient for most mobile agent applications. In addition, the code of system messengers are cached in every IMAGO server. As a consequence, migration of a system messenger only involves moving its message and run time stack. We call this technique as zero-code migration which greatly reduces the overhead of agent migration. For the sake of flexibility, we also allow users to define their application-specific messengers. However, user designed messengers can be dispatched from the home server only, because the IMAGO system has to load the user messenger code from the local secondary storage while file access is strictly forbidden on any remote server for security reason.

The dispatcher of a messenger only specifies who is the receiver and the message to be delivered, it is the messenger's responsibility to locate the receiving agent. In general, the receiver may be a single worker name, or a non-empty list of worker names such as demonstrated by the *$multicasting_messenger* and the *$postman_messenger*.

In IMAGO Prolog, messages are represented purely by Prolog itself. In other words, the language element specifies that the message content is expressed as a Prolog term which could be an atom, a variable, a list, or a compound structure. Compare with other ACLs, performatives of messages are implicitly determined by the messenger type. For example, a messenger created from *$cod_messenger* or *$collecting_messenger* delivers a QUERY-IF message while a messenger created from other messenger prototypes carries a message with a default performative INFORM.

Obviously, the performative and content of a message often determine the reaction of the receiver. In addition to the various types of system messengers for sending agents, the IMAGO Prolog provides a set of builtin predicates for receiving agents. The predicate which is similar to an unblocking receive is *accept(Sender, Msg)*. An invocation to this predicate succeeds if a matching messenger is found, or fails if either the caller's messenger queue is empty or there is no matching messenger in the queue. Likewise, the predicate which implements blocking receive is *wait_accept(Sender, Msg)*. A call to this predicate succeeds immediately if a matching messenger is found. However, it will cause its caller to be blocked if either the caller's messenger queue is empty, or no matching messenger can be found. In this case, it will be automatically re-executed when a new messenger attaches to the caller's messenger queue.

Pragmatically, the semantics of *matching messenger* is implemented by Prolog unification. Let *(S, M)* be the sender and content element carried by a messenger, and *(Sender, Msg)* be the arguments of an accept-like primitive, the messenger is a matching messenger to the accept-like primitive if the general unification of *(S, M)* and *(Sender, Msg)* succeeds. A messenger attached to an imago is ready to be searched by *accept/2* or *wait_accept/2*. The behavior of an accepting predicate is determined by Table 2: it succeeds if one of the cases in the table is satisfied, or it fails/waits otherwise. A failed accepting predicate does not cause any side effect and the messenger queue remains unchanged.

Table 2. Behavior of Accepting Predicates

Sender	*Msg*	*Behavior*
var	var	the 1st message
var	nonvar	the 1st message unifying with *Msg*
nonvar	var	the 1st message unifying with *Sender*
nonvar	nonvar	the 1st message unifying with both *Msg* and *Sender*

For example, suppose worker *alice* executes the following code:

```
current_host(H),
Msg = exchange(H, Y),
dispatch($cod_messenger, white_rabbit, Msg),
wait_accept(white_rabbit, Msg),
show(Y),
...
```

and worker *white_rabbit* evaluates:

```
current_host(H),
Msg = exchange(X, H),
wait_accept(alice, Msg),
show(X),
...
```

The behavior of the execution looks like a synchronous remote unification. By dispatching one *$cod_messenger*, both *alice* and *white_tabbit* finally receive the same *Msg* which make them to exchange their host addresses.

5 Communication Patterns

The concept of messengers offers flexibility to simulate different patterns of agent communication. Users not only have choices to dispatch different messengers for various purposes, but also have the freedom to accept messengers either selectively or in some priority order (based on the binding of the sender's name and the message). In this section, we show a few possible communication patterns.

First-Come-First-Serve Receiving:

```
w(...) :-
        wait_accept(Who, Msg),
        // processing Msg
        w(...).
```

Selective Receiving:

```
w(...) :-
        wait, // wait for a new message
        w1(...).
w1(...) :-
        accept(Who, selected(M)), !,
        // process selected(M) from Who
        w(...).
w1(...) :-
        accept(my_buddy, Msg), !,
        // process Msg from my_buddy
        w(...).
w1(...) :-
        accept(W, M), !,
        // process any message M from any sender W
        w(...).
```

Ordered Receiving:

```
    w(...) :-
            wait_accept(W, first(X)),
            // process the message unifies with first(X)
            wait_accept(W, second(Y)),
            // process the message unifies with second(Y)
            other_processing(...).
```

To illustrate how the IMAGO Prolog semantics is actually satisfied during inter-agent communication, let us take a look at another example. Suppose that three worker instances, namely, a *producer*, a *consumer* and an unbounded buffer *stack* are created from the following agent prototypes respectively.

```
:- worker(ima_producer).
    ima_producer(Stack)) :-
            produce(X),
            dispatch($oneway_messenger, Stack, push(X)),
            ima_producer(Stack).
    produce(X) :-
            // code to produce X.
:- end_worker(ima_producer).

:- worker(ima_consumer).
    ima_consumer(Stack) :-
            dispatch($cod_messenger, stack, pop(X)),
            wait_accept(stack, pop(X)),
            consume(X),
            ima_consumer.
    consume(X) :-
            // code to consume X.
:- end_worker(ima_consumer).

:- worker(ima_stack).
    ima_stack([]) :-
            wait_accept(_, push(X)), !,
            ima_stack([X]).
    ima_stack([X|L]) :- accept(_, pop(X)), !,
            ima_stack(L).
    ima_stack(L) :- accept(_, push(X)), !,
            ima_stack([X|L]).
    ima_stack(L) :- wait,
            ima_stack(L).
:- end_worker(stack).
```

Let us now take a closer look at the behavior of different agents in this example. The *producer* agent repeatedly generates an item X and then dispatches an *$oneway_messenger* to deliver a push(X) message to the *stack* agent. On the other hand, the *consumer* agent repeatedly dispatches an *$cod_messenger* to the *stack* agent requesting a pop(X) operation, waits for the messenger coming back with an instantiated X, and then consumes X.

To prevent stack underflow, the *stack* agent only accepts a messenger who carries a push(X) message if the stack is empty. Otherwise, it tries to accept any available messenger and takes the corresponding action. If there is no messenger in its current messenger queue, the *stack* agent calls *wait* which blocks the caller and resumes it as soon as a new messenger arrives. As we mentioned before, the purpose of an *$oneway_messenger* is to inform that the content of its message is a *true* proposition. Thus the *stack* agent simply accepts this proposition, *i.e.*, the *push(X)*. However, the purpose of an *$cod_messenger* is to query if the content of its message is a *true* proposition. It is the *$cod_messenger*'s responsibility to inform its sender when the query, *i.e.*, the *pop(X)*, is evaluated. In order to match a query-type messenger, unification must be carried in both sides, namely, one in *stack*'s working space and another in the messenger's working space.

Even though IMAGO Prolog aims at mobile agent applications, the agent-based communication model can be used to simulate remote procedure calls in the client/server model as well. For example:

```
:- worker(server).
     server :-
             wait_accept(W, Goal),
             clone(_, R),
             R == clone →
               call(Goal),
               dispatch($oneway_messenger, W, Goal),
               dispose;
               server.

     // definitions of procedures
     ....
:- end_worker(server).
```

In essence, the above code segment gives a concurrent server organization, *i.e.*, the server does not handle the RPC request itself, but passes the RPC to a separate worker, after which it immediately waits for the next incoming RPC. More precisely, when a *server* agent is started, it waits for an incoming messenger and clones itself as soon as an RPC (represented by *Goal*) is received. After cloning, the original worker loops back to wait and accept the next messenger, whereas the cloned worker issues a *call(Goal)* to execute the procedure, dispatches a messenger back to the caller to report execution result, and then disposes itself. It is worthwhile to note that the *clone* predicate is implemented in an efficient way in IMAGO Prolog, we only need to allocate an execution stack to the cloned worker and let it share the same code segment as its cloner.

The advantage of our Prolog-based ACL is that inter-agent communication fulfills the semantics defined by the language (Prolog) of the content expression.

6 Conclusion

In this paper, we discussed different communication concepts for inter-agent communication, and investigated these concepts with respect to existing mobile agent systems. The major concern of our discussion has focused on the communication models and agent communication languages. Based on the survey, we presented an agent-based model which deploys intelligent mobile messengers for inter-agent communication. We have successfully implemented the IMAGO Prolog as well as its execution platform. An evaluation release of the IMAGO system is available on the IMAGO Lab web site.

The advantage of our agent-based model is that a simple, reliable agent migration protocol is sufficient to support both agent migration and inter-agent communication. Moreover, we provided a high level programming abstraction which makes the ACL and the agent programming language integrated within the same framework - the IMAGO Prolog. The advantage of our Prolog-based ACL is that inter-agent communication fulfills the semantics defined by the language (Prolog) of the content expression. However, besides this contribution, our proposed Prolog-based ACL is more limited than existing ACLs (FIPA ACL particularly). For example, it is not possible to define protocols and follow the flow of a conversation. In addition, performatives in our ACL are implicit which might limit the expressive power in inter-agent communication.

Although this study concentrates on the design of inter-agent communication mechanism in mobile agent systems, results will be also useful in related disciplines of network/mobile computing and logic programming community. Finally, we would like to express my appreciation to the Natural Science and Engineering Council of Canada for supporting this research.

References

1. D. B. Lange and M. Oshima, "Programming and Deploying Java Mobile Agents with Aglets", *Addison-Wesley*, August, 1998.
2. J. Baumann *et al.*, "Mole - Concepts of Mobile Agent System", *World Wide Web*,(1)3, 1998, pp. 123-137.
3. H. Kopetz *et al.*, "Agent Tcl: Targeting the Needs of Mobile Computers", *IEEE Internet Computing*, (1)4, 1997, pp. 58-67.
4. "ObjectSpace: ObjectSpace Voyager Core Package Technical Overview", *Technical Report, ObjectSpace Inc.*, 1997, http://www.objectspace.com/.
5. A. Suna and A. EI Fallah-Seghrouchni, "A Mobile Agents Platform: Architecture, Mobility and Security Elements", *in Proceedings of ProMAS'04, Workshop of AAMAS*, New York, USA, 2004, pp. 57-66
6. A. EI Fallah-Seghrouchni and A. Suna, "CLAIM: A Computational Language for Autonomous, Intelligent and Mobile Agents", *in Proceedings of ProMAS'03, Workshop of AAMAS*, Melbourne, Astralia, 2003
7. H. Peine, "Ara - Agents for Remote Action", *in Mobile Agents: Explanations and Examples (eds. W. Cockayne and M. Zyda)*, Manning/Prentice Hall, 1997
8. "Concordia", *Mitsubishi Electric*, http://www.meitca.com/HSL/Projects/Concordia.

9. E. Pitoura and G. Samaras, "Locating Objects in Mobile Computing", *IEEE Trans. on Knowledge and Data Engineering*, (13)4, 2001, pp. 571-592.
10. A. Tanenbaum and M. van Steen, "Distributed Systems", *Prentice Hall, Inc.*, 2002
11. A. Murphy and G. Picco "Reliable Communication for Highly Mobile Agents", *in Proceedings of ASA/MA'99*, 1999, pp. 141-150.
12. J. Cao *et al.*, "Mailbox-Based Scheme for Mobile Agent Communications", *IEEE Computer*, (35)9, 2002, pp. 54-60.
13. T. Finin *et al.*, "KQML as an Agent Communication Language", *CIKM'94*, ACM Press, 1994, pp. 456-463.
14. "Foundation for Intelligent Physical Agents - FIPA'99 Version 0.2", *FIPA*, http://www.fipa.org/.
15. X. Li, "IMAGO: A Prolog-based System for Intelligent Mobile Agents", *in Proceedings of Mobile Agents for Telecommunication Applications (MATA'01), LNCS 2164*, Springer-Verlag, 2001, pp. 21-30.

OMNI: Introducing Social Structure, Norms and Ontologies into Agent Organizations

Virginia Dignum, Javier Vázquez-Salceda, and Frank Dignum

Institute of Information and Computing Sciences,
Utrecht University, The Netherlands
{virginia, javier, dignum}@cs.uu.nl

Abstract. In this paper, we propose a framework for modelling agent organizations, Omni, that allows the balance of global organizational requirements with the autonomy of individual agents. It specifies global goals of the system independently from those of the specific agents that populate the system. Both the norms that regulate interaction between agents, as well as the contextual meaning of those interactions are important aspects when specifying the organizational structure. Omni integrates all this aspects in one framework. In order to make design of the multi-agent system manageable, we distinguish three levels of abstraction with increasing implementation detail. All dimensions of Omni have a formal logical semantics, which ensures consistency and possibility of verification of the different aspects of the system. Omni is therefore utmost suitable for the modelling of all types of MAS from open to closed environments.

1 Introduction

In closed domains, the design of MAS can suffice with the idea that agents are mere performers of organizational roles or functions, interacting according to fixed protocols and unable to deviate from expected behavior [21]. As such, agent autonomy is rather limited. In open domains, agents are self-governed autonomous entities that pursue their own individual goals based only on their own beliefs and capabilities [1].

Comprehensive models for MAS must, on the one hand, be able to specify global goals and requirements of organizations but, on the other hand, cannot assume that participating agents will act according to the needs and expectations of the system design. Concepts as organizational rules [20], norms and institutions [6], [7], and social structures [13] arise from the idea that the effective engineering of MAS needs high-level, agent-independent concepts and abstractions that explicitly define the organization in which agents live [21]. These are the rules and global objectives that govern the activity of an enterprise, group, organization or nation.

Given that agents might deviate from expected behavior, open societies need mechanisms to systematize, defend and recommend right and wrong behavior, which can inspire trust into the agents that will join them. Norms are commonly used means to describe such expected behavior. Finally, organizational models must provide means to represent concepts and relationships in the domain that are rich enough to *cover* the necessary contexts of agent interaction while keeping in mind the *relevance* of those concepts for the global aims of the system.

In this paper we propose a framework for agent organizations, Omni(Organizational Model for Normative Institutions) presenting a first attempt to cover all the above mentioned aspects in a way that is usable for both open and closed systems.

The paper is organized as follows: In §2 we present a generic description of the Omni framework. In §3, we discuss the abstract level of an organization. Then we will focus on the description of the Organizational dimension (§4), the Normative dimension (§5) of the e-Organization and present an outline in §6 of the kind of ontologies and communication languages needed in the Ontological Dimension. In §7 we compare our framework with other well known models. We end this document with our conclusions and outline future lines of research. Throughout the paper the different components of a society are illustrated using an organization that has as main global objective the realization of conferences.

2 The Omni Framework

Omni is an integrated framework for modelling a whole range of MAS, from closed systems with fixed participants and interaction protocols, to open, flexible systems that allow and adapt to the participation of heterogeneous agents with different agendas. This approach is rather unique, as most existing frameworks concentrate in a specific type of MAS. Omni is composed by three dimensions: **Normative**, **Organizational** and **Ontological** that describe different characterizations of the environment. The model is based on two recent MAS models, OperA [5], and Harmon*IA* [18]. Figure 1 depicts the different modules that compose our proposed framework organized into three levels of abstraction:

– the **Abstract Level**: where the statutes of the organization to be modelled are defined in a high level of abstraction. This step is similar to a first step in the requirement analysis. It also contains the definition of terms that are generic for any organization (that is, that are incontextual) and the ontology of the model itself.
– the **Concrete Level**: where all the analysis and design process is carried on, starting from the abstract values defined in the previous level, refining their meaning in terms of norms and rules, roles, landmarks and concrete ontological concepts.
– the **Implementation Level**: where the design in the Normative and Organizational dimensions is implemented in a given multi-agent architecture, including the mechanisms for role enactment and for norm enforcement.

The division of the system into these three levels aims to ease the transition from the very abstract statutes, norms and regulations to the very concrete

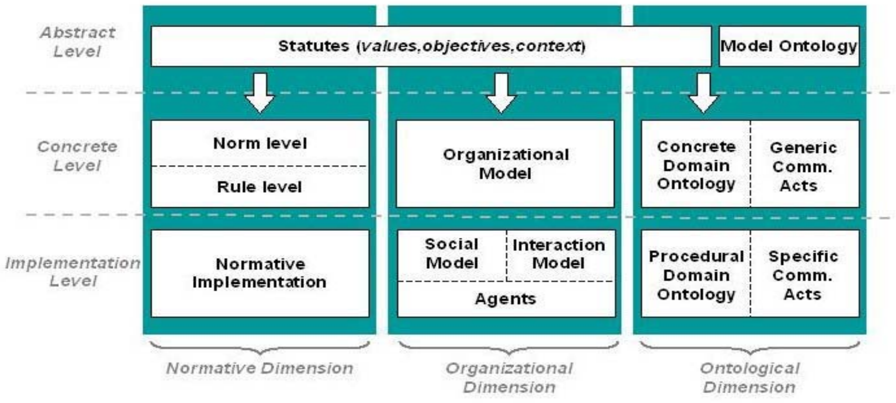


Fig. 1. The OMNI framework

protocols and procedures implemented in the system. Different domains have different requirements concerning normative, organizational and communicative characteristics, which means that not always all three modules have the same impact or are even needed: in those domains with none or small normative components, design is mainly guided by the Organizational Dimension, while in highly regulated domains the Normative Dimension is the most prominent.

3 The Abstract Level

Statutes indicate, at the most abstract level, the main *objective* of the organization, the *values* that direct the fulfilling of this objective and they also point to the *context* where the organization will have to perform its activities.

In the conference scenario, we can take as example a research consortium such as the IFMAS (International Foundation for Multi-Agent Systems). The statutes state: *"IFMAS is a non-profit corporation whose purpose is to promote science and technology. In pursuit of its purposes, IFMAS will engage in activities including, but not limited to: (1) Coordinating and arranging seminars on artificial intelligence and multi-agent systems; (2)..."*. In this statement we can find:

1. the *objectives*: the main objective of this organization is to promote science and technology. Another objective is the organization of seminars.
2. the *context*: IFMAS states that it operates in the area of artificial intelligence and multi-agent systems.
3. the *values*: The IFMAS is a *non-profit* organization. Implicit in the latter part, it also says that sharing is also a value of the organization.

The *objectives* of the organization express the overall goals of the society. As far as the organization has control over the actions of the agents acting within that organization, it will try to ensure that they perform actions leading to the overall goal of the society. We will see in §4.1 and §4.2 how these objectives influence the design process in the *Organizational Dimension*.

The *values* of the organization are beliefs that individuals have about *what* is important, both to them and to the society as a whole. A value, therefore, is a belief (right or wrong) about the way something should be. Values define beliefs about, for instance, what is acceptable or unacceptable, good or bad. In our framework, values are the basis of the *Normative Dimension*.

The environment, or *context*, of an organization can be seen as consisting essentially of other societies or organizations that are interdependent and each influence the other.

4 The Organizational Dimension

The design of agent organizations must capture on the one hand, the structure and requirements of the society owners, and on the other hand, must assume that participating agents must be available that are able and interested in enacting society roles. In OMNI, the Organizational Dimension consists of a 3-layered model: based on the concerns identified in the Abstract Level, the Concrete Level specifies the structure and objectives of a system as envisioned by the organization, and the Implementation Level describes the activity of the system as realized by the individual agents. This separation enables OMNI models to respect the autonomy of individual agents while ensuring conformance to organizational aims.

4.1 The Organizational Abstract Level

The abstract level of the Organizational Dimension describes which are the aims and concerns of the organization with respect to the social system. At the abstract level, as we saw in §3, this is defined by means of a list of the organization's externally observable *objectives*, that is, the desired states of affairs in the life of the society. These abstract objectives are translated into the specific objectives of the society model. In the case of our example, the abstract objective of *"coordinating and arranging seminars..."* is translated into the objective of organizing a specific conference.

A common way to express the objectives of an organization is in terms of its expected functionality. The determination of the overall objectives of the society follows a process of elicitation of functional (*what*) and interaction (*how*) requirements. For example, how should a conference be organized, in terms of program, location, co-located workshops, etc. To identify the objectives of an organization, it is important to characterize the different stakeholders (*who*) of the organization, their requirements, expectations, constraints and relationships to each other. Stakeholders in the conference scenario are researchers, organizers, etc. Stake holders are the basis for the identification of *roles* in the concrete level of specification of an organization (see §4.2).

4.2 The Organizational Concrete Level

The Concrete Level of the organizational dimension specifies the *means* to achieve the objectives identified in the the abstract level as an *Organizational Model* (OM). The OM describes the structure and global characteristics of a domain from an organizational perspective. That is, from the premise that it is the society goals that determine roles and interaction norms. Organizational characteristics of an agent society are specified in the OM in terms of its *Social* (§4.2) and *Interaction Structures* (§4.2).

The definition of these structures alone is not enough, as specification of a society should include the description of concepts and relations holding in the domain, and of those behaviours accepted as 'good'. Therefore, these structures should be linked with the *role norms*, *scene norms* and *transition norms*, defined in the concrete level of the Normative Dimension (see §5.2), and with the ontologies and communication languages defined in the concrete level of the Ontological Dimension (see §6).

The organization design is also guided by the coordination needs of the domain. These determine the type of roles and tasks necessary to facilitate the tasks of the organization. We identify three basic coordination types: *market*, *hierarchy* and *network* that are defined as *Architectural Templates.*

The Social Structure. The social structure of an organization describes the roles holding in the organization. It consists of a list of role definitions, *Roles* (including their objectives, rights and requirements), such as PC-member, program chair, author, etc.; a list of role groups' definitions, *Groups*; the relations between roles by a *Role Hierarchy* graph, and a *Role Dependencies* graph.

Roles are the main element of the *Social Structure.* From the society perspective, role descriptions should identify the activities and services necessary to achieve society objectives and enable to abstract from the individuals that will eventually perform the role. From the agent perspective, roles specify the expectations of the society with respect to the agent's activity in the society. In Omni, the definition of a role consists of an identifier, a set of role objectives, possibly sets of sub-objectives per objective, a set of role rights, a set of norms and the type of role. An example of role description is presented in table 1. The meaning and relationships between the concepts used is formally specified in the Ontological Dimension, and the formal specification of norms in the Normative Dimension.

Groups provide means to collectively refer to a set of roles. Moreover, *groups* are used to specify norms that hold for all roles in the group. Groups are defined by means of an identifier, a non-empty set of roles, and the group norms. Norms of a group must be consistent with the norms of the roles in the group. An example of a group in the conference scenario is the organizing team consisting of the roles *program chair*, *local organizer*, and *general chair.*

Abstract society objectives form the basis for the definition of the objectives of roles. The distribution of objectives in roles is defined by means of the *Role Hierarchy.* Different criteria can guide the definition of *Role Hierarchy.* In par-

Table 1. *PC member* role description

Id	PC_member
Objectives	paper_reviewed(Paper,Report)
Sub-objectives	{read(P), report_written(P, Rep), review_received(Org, P, Rep)}
Rights	access-confmanager-program(*me*)
Norms & Rules	PC_member is OBLIGED to understand English IF paper_assigned THEN PC_member is OBLIGED to review paper BEFORE given deadline IF author of paper_assigned is colleague THEN PC_member is OBLIGED to refuse to review ASAP
Type	external

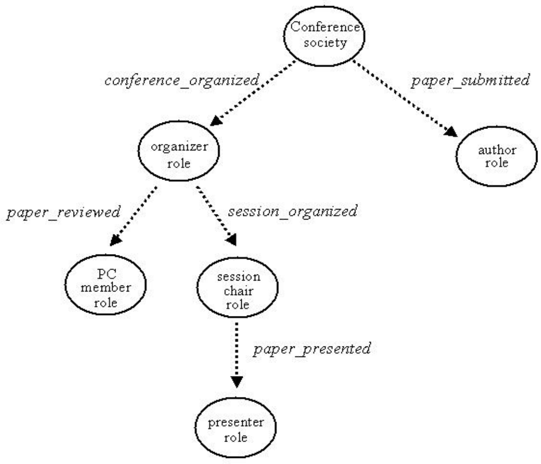


Fig. 2. Role dependencies in a conference

ticular, a role can be refined by decomposing it in sub-roles that, together, fulfill the objectives of the given role. This refinement of roles defines *Role Dependencies*. A dependency graph represents the dependency relations between roles. Nodes in the graph are roles in the society. Arcs are labelled with the objectives of the parent role for whose realization the parent role depends on the child role. Part of the dependency graph for the conference society is displayed in figure 2. For example, the arc between nodes PC-Chair and PC-member represents the dependency *PC-Chair* $\succeq_{paper-reviewed}$ *PC-member*. The way the objective g in a dependency relation $r_1 \succeq_g r_2$ is actually passed between $r1$ and $r2$ depends on the coordination type of the society, defined in the Architectural Templates. In OMNI, we identify three types of role dependencies: *bidding*, *request* and *delegation*.

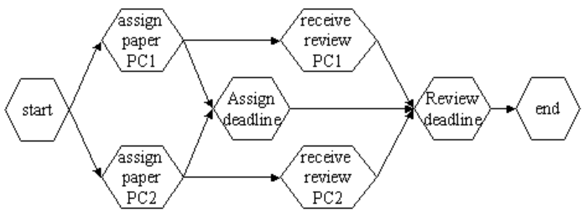


Fig. 3. Landmarks pattern for review process

The Interaction Structure. Interaction is structured in a set of meaningful scenes that follow pre-defined abstract scene scripts. Examples of scenes are the registration of participants in a conference, which involves a representative of the organization and a potential participant, or paper review, involving program committee members and the PC chair. A *scene script* describes an scene by its players (roles), its desired results and the norms regulating the interaction. In the OM, scene scripts are specified according to the requirements of the society. The results of an interaction scene are achieved by the joint activity of the participating roles, through the realization of (sub-)objectives of those roles. A scene script establishes also the desired *interaction patterns* between roles, that is, a desired combination of the (sub-) objectives of the roles.

Table 2. Script for the *Review Process* scene

Scene	Review Process
Roles	Program-Chair (1), PC-member(2..Max)
Results	$r_1 = \forall$ P $\in$ Papers: reviews_done(P, review1, review2)
Interaction Patterns	PATTERN(r_1): *see figure 3*
Norms & Rules	Program-Chair is PERMITTED to assign papers PC-member is PERMITTED to review papers assigned before deadline

In OMNI, interaction description is declarative in nature, rather than describing the exact activities. Interaction objectives can be more or less restrictive, giving the agent enacting the role more or less freedom to decide how to achieve the role objectives and interpret its norms. Following the ideas of [15], we call such expressions *landmarks*, that is, conjunctions of logical expressions that are true in a state. Figure 3 shows the informal landmark pattern for the *Review Process.*

Several different specific actions can bring about the same state, that is, landmarks actually represent families of protocols. The use of landmarks to describe activity enables the actors to choose the best applicable actions, according to their own goals and capabilities.

The relation between scenes is represented by the *Interaction Structure* (see figure 4). In this diagram, *transitions* describe a partial ordering of the scenes,

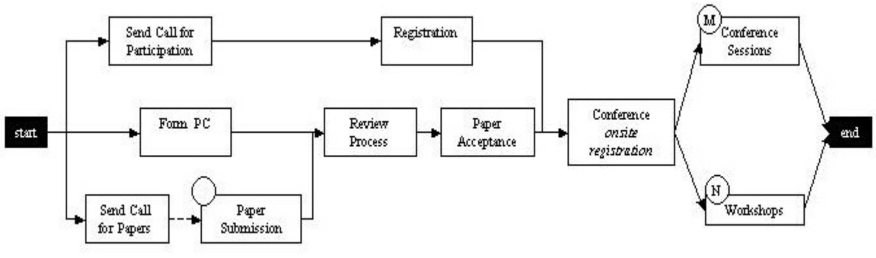


Fig. 4. Interaction Structure in the Conference scenario

plus eventual synchronization constraints. Note that several scenes can be happening at the same time and one agent can participate in different scenes simultaneously. Transitions also describe the conditions for the creation of a new instance of the scene, and specify the maximum number of scene instances that are allowed simultaneously. Furthermore, the enactment of a role in a scene may have consequences in following scenes. In these cases the *evolution relations* between roles describe the constraints that hold for the role-enacting agents as they move from scene to scene.

4.3 The Organizational Implementation Level

Omni assumes that individual agents are designed independently from the society to model goals and capabilities of a given entity. Individual agents will join a society as enactors of organizational role(s), as a means to realize their own goals [3].

Social Model. Agent populations of the organizational model are described in the social model (SM) in terms of commitments regulating the enactment of roles by individual agents. Depending of the specific agents that will join the organization, several populations are possible for each organizational model.

When an agent applies, and is accepted, for a role, it will commit itself to the realization of the role goals and to act within the society according to the role constraints. The commitments are specified as social contracts. A *social contract* describes the conditions and rules applying to an agent enacting role(s) in the agent society. Given an agent society S, a social contract for agent s enacting role r is defined as a tuple

$$\textit{social-contract} = \langle a, r, CC \rangle$$

where a is an agent, $r \in \textit{roles}(S)$ is a role, and CC is a set of contract clauses (including (1) the time period the contract holds -either from date to date, or until certain states hold; (2) specific agreements and conditions governing the role enactment, and (3) the sanctions to take when norms are violated). In the conference scenario, when a researcher becomes PC member, her social contract will describe for example the agreements concerning number of papers to review (which possibly may deviate from the standard number desired by the conference).

Social contracts identify *role enacting agents* (*reas*) that compose the society. For each agent, the *rea* reflects the agent's own requirements and conditions concerning its participation in the society. Making agreements explicit and formal, allows the verification of whether the animated society behaves according to the design specified in the OM. Social contracts in OMNI are a two-sided agreement between agents and roles instead of a one-sided API description of role enactment, as have been proposed by other researchers [16, 12]. In the extreme, if all is expressed in the role definition and no room is left for negotiation, OMNI social contracts can function as these API's.

Interaction Model. OMNI provides two levels of specification for interactions. While the OM provides a script for interaction scenes according to the organizational aims and requirements, the IM, realized in the form of contracts, provides the interaction scenes such as agreed upon by the agents. Due to the autonomous behavior of agents, the interaction model must be able to accommodate other interaction contracts describing new, emergent, interaction paths, to the extent allowed by the organizational and social models.

An interaction scene results from the instantiation of a scene script, described in the OM, to the reas actually enacting it and might include specializations or restrictions of the script to the requirements of the reas. An interaction contract describes the conditions and rules applying to interaction between agents in the agent society. That is, the clauses in an interaction contract specify actual instantiations of interaction scene scripts and must indicate the actors involved and the specific agreements and sanctions concerning the scene to be played. The contract must furthermore involve sufficient reas to cover all the needed roles in the scene. Besides the refinement of the script to the desires and characteristics of the agents participating in the scene instance, interaction contracts must describe the protocol agreed by those agents to fulfil the script landmarks. Interaction protocols are the concrete representation of the refinement of scene script landmarks with the particularities imposed by the participants to the specific communicative capabilities of those participants. Given a society S and a scene $s \in scenes(S)$, an *interaction contract* is defined as a tuple

$$interaction\text{-}contract = \langle A, s, CC, P \rangle$$

where the set of agents $A = \{a \in Agents : rea(a, r, s) | r \in roles(s)\}$ represents the set of all agents enacting reas participating in interaction scene s, CC is a set of contract clauses, that is, possible conditions and deadlines concerning the results and interaction patterns of scene s, and P is the protocol to be followed. Protocols describe the actual interaction between reas. A rea interaction protocol describes a communication pattern for reas that fulfills the scene script landmarks. In the conference scenario, an interaction contract for the Review Process scene can specify, for example, that actors will follow the *ConfMaster* protocol.

5 The Normative Dimension

In the same way as the Organizational Dimension, the Normative Dimension is composed by the different levels of abstraction. The translation steps from one level to the following are described in a formal way, as we aim to to be able to verify if a given organization complies to all the norms that are specified in the regulations. The connection between levels is very useful not only in its *top-down* direction (guiding the design), but also from *bottom up* (agents can trace the origin of a given protocol and reason in terms of the rules and norms the protocol implements).

5.1 The Normative Abstract Level

The values and objectives of an organization described in the Abstract Level, can be described as the desires of the society model. For example, for the Conference Society :

- the *information sharing* value can be described as $D(share(info))$,
- the *non-profit* value can be described as $D(non\text{-}profit(organization))$,

However, besides a formal syntax, this does not provide any meaning to the concept of *value*. That is, values do not specify *how*, *when* or in *which* conditions individuals should behave appropriately in any given social setup. This will be defined later by abstract norms, concrete norms and rules (see section 5.2). In OMNIthe meaning of the values is defined by the norms that contribute to this value. In an intuitive way we can see this translation process as follows:

$$\vdash_{org} D(\varphi) \mapsto O_{org}(\varphi)$$

meaning that, if an organization *org* values situations where φ holds higher than situation where φ does not hold, then such value can be translated in terms of a norm (an obligation of the organization *org*) to fulfill φ. In our framework a norm *contributes to a value* if fulfilling the norm always leads to states in which the value is more fully accomplished. So, each value has attached a list of norms that contribute to that value.

$$\vdash_{IFMAS} D(share(info)) \mapsto O_{IFMAS}(disseminate(research))$$

We define *ANorms* (the language for abstract norms) to be a deontic logic that is temporal, relativized (in terms of roles and groups) and conditional, i.e., an obligation to perform an action or reach a state can be conditional on some state of affairs to hold, it is also meant for a certain type (or role) of agents and should be fulfilled before a certain point in time. For instance, the following norm might hold: *"The authors should submit their contributions before the deadline"*, which can be formalized as:

$$O_{author}(submit(paper) < Deadline)$$

The obligation is directed towards the author, assuming that she is responsible for fulfilling it.

5.2 The Normative Concrete Level

In order to check norms and act on possible violations of the norms by the agents within an organization, the abstract norms have to be translated into actions and concepts that can be handled within such organization. To do so, the definition of the abstract norms are iteratively concretized into more concrete norms, and then translated into the rules, violations and sanctions that implement them.

The Norm Level. The norms at this level are described in $CNorms$ (the language for concrete norms), which we assume for the moment to be equal to $ANorms$, but which might use different predicates. In addition we define a function `I`: $ANorms \rightarrow CNorms$ which is a mapping from the abstract norms to the concrete ones. For each abstract norm `I` indicates how it can be fulfilled by fulfilling concrete norms within the context of this organization. This function is based on the counts-as operator as developed in [8].

There are several ways in which norms can be *abstract* and thus several ways to make them more concrete [18]. As an illustration of this process, in the following we describe two kinds of abstractness.

Abstract actions: Actions that can be implemented in many ways. For example: *"submitting a paper"*. The translation in this case is a kind of definition of the abstract action in terms of concrete ones:

$$\begin{array}{l} \mathit{send\_mail}(\mathit{organizer,\ files}) \cup \mathit{send\_post}(\mathit{organizer,\ hard\_copies}) \\ \Rrightarrow_{IFMAS}\ \mathit{submit}(\mathit{paper}) \end{array}$$

Temporal abstractness: Often there is an implicit deadline for obligations. E.g., the obligation of reviewing the paper occurs only if the paper is assigned, and if so the review should be done before the deadline:

$$\begin{array}{l} \mathit{done}(\mathit{assign\_paper}(P,me,Deadline)) \rightarrow \\ O_{PC\_member}(\mathit{review\_paper}(P,\ Rep) < \mathit{do}(\mathit{pass}(Deadline))) \end{array}$$

The Rule Level. The translation from norms to rules in OMNI marks a transition from a normative perspective to a more descriptive one. Such translation also implies a change in the language, from deontic logic to a Propositional Dynamic Logic (a language more suitable to express actions and time constraints). Each norm can be translated into:

a) *a violation expression*: by using the following reduction rule by Meyer [10]:

$$O(\alpha) \mapsto [\neg\alpha]\, V$$

b) *a precedence expression*: in those cases where the norm expresses temporal relations among actions, such relation can be also expressed through the [ ] operator as follows:

$$\begin{array}{l} O(\alpha < \mathit{do}(\beta)) \mapsto [\beta]\, \mathit{done}(\alpha) \\ O(\alpha < \mathit{do}(\beta)) \mapsto \neg \mathit{done}(\alpha) \rightarrow [\beta]\, V \end{array}$$

The first reduction rule translates the temporal constraint of α being done before β with an expression in Dynamic Logic that states: *"once action β is performed, it should always be the case that action α has been done"*. The second reduction rule expresses the violation condition: *"if action α has not been done, once action β is performed it always is the case that violation V occurs"*.

Table 3. Formal specification of *PC member* role

Id	PC_member
Objectives	paper_reviewed(Paper,Report)
Sub-objectives	{read(P), reported(P, Rep), review_received(Org, P, Rep)}
Rights	access-confman-program(*me*)
Norms	O_{PC_member}*(understand(English))* *done(assign_paper(P,me,Deadline))* $\rightarrow$ O_{PC_member}*(review_paper(P, Rep) < do(pass(Deadline)))* *done(assign_paper(P,me,_))* $\wedge$ *is_a_direct_colleague(author(P))* $\rightarrow$ O_{PC_member}*(review_refused(P) < pass(TOMORROW))*

Rules	*done(assign_paper(P,me,Deadline))* $\wedge$ $\neg$*done(review_paper(P, Rep))*
	$\rightarrow$ [*pass(Deadline)*] *V4*
	done(assign_paper(P,me,_)) $\wedge$ *is_a_direct_colleague(author(P))* $\wedge$
	$\neg$*done(review_refused(P))* $\rightarrow$ [*pass(TOMORROW)*] *V5*

Type	external

By means of this refinement process, the designer can obtain all the norms and rules that apply in the system, and then include them in the organizational model. An example of the formalization of the role norms introduced in table 1 is given in table 3.

In OMNI, *violations* are the key concept in norm enforcement. We separate the violations coming from the behavior of external entities (*external violations*) from the ones related to the behavior of the internal agents (*internal violations*)

Internal violations describe states that the organization should always avoid. As the designer has full control of the design of the organization's own agents, such internal agents will fully comply with the organizational objectives and follow its norms and rules. In this case the aim is not to create an *enforcement mechanism* but a continuous *safety control* of the system's behavior (i.e., avoid the system to enter in a non-desirable, *illegal* state because of a failure in one of the agents).

In our framework, *external violations* are the ones where the designer should pay more attention. As we cannot assume that agents entering into the organization will always follow the norms and rules imposed by its normative system, an active enforcement should be made by the internal agents. In our framework, internal agents do not have access to the internal beliefs, goals and intentions of the other agents, they can only check the agents' behavior, by detecting when those agents enter in states considered *illegal*. The way of doing so is by means of the list of definitions of external violations. Such list defines, for each violation, the condition that triggers it. This condition is extracted from the rule that defines the violation. As an example, let us take one of the rules identified for the PC member role in table 3:

$$done(assign\_paper(P,me,Deadline)) \\ \wedge \neg done(review\_paper(P,Rep)) \rightarrow [pass(Deadline)] \text{ V4}$$

we can create the condition for violation $V4$ by stating that the action inside [ ] has been *done* while the other preconditions are true. Then we should also add the *sanction* (the actions carried against the violator), the *side-effects* (the actions to be done to counter-act the violation) and the *enforcing roles* (the role or roles that have the responsibility to detect this type of violations):

`Violation:`	*IFMAS:V4*
`Pre-conditions:`	$done(assign\_paper(P,me,Deadline))$
	$\wedge \neg\ done(review\_paper(P,\ Rep)) \wedge done(pass(Deadline))$
`Sanction:`	$delete\_from\_PC\_list(me))$
`Side-effects:`	$\{\ find\_new\_reviewer(P,r2);\ assign\_paper(P,r2,Deadline2)\ \}$
`Enforcing roles:`	$\{\ organizer,\ session\_chair\}$

5.3 The Normative Implementation Level

There are two main approaches to implement the rules in the rule level: a) creating a *rule interpreter* that any agent entering the organization will incorporate, and b) translating the rules into *protocols* to be included in the interaction contracts

Note that in both cases it is not ensured that the agents will follow those descriptions. The violations in the rule level should also be translated in some detection mechanisms to check the behavior of the agents.

At Implementation Level, the organization model provides both the low-level protocols and the related rules. Agents that are only able to follow protocols will blindly follow them, while the ones that can also interpret the rules (*Deliberative Normative Agents* [4]) can choose between following the protocol or reasoning about the rules, or do both. With this approach the autonomy of the agents entering the organization is adapted to their reasoning capabilities, which makes OMNI utmost adequate to model open environments. *Norm Autonomous Agents* that are able to reason about norms, can switch from following low-level protocols to higher level rules and norms, by using the links provided by OMNI from procedures to rules, and from rules to norms. An example is shown in figure 5.

6 The Ontological Dimension

The main challenge of coordination and collaboration in open environments is that of mutual understanding. Communication mechanisms include both the representation of domain knowledge (*what* are we talking about) and protocols for communication (*how* are we talking). Both content and protocol have different meanings at the different levels of abstraction (e.g. while at the abstract level

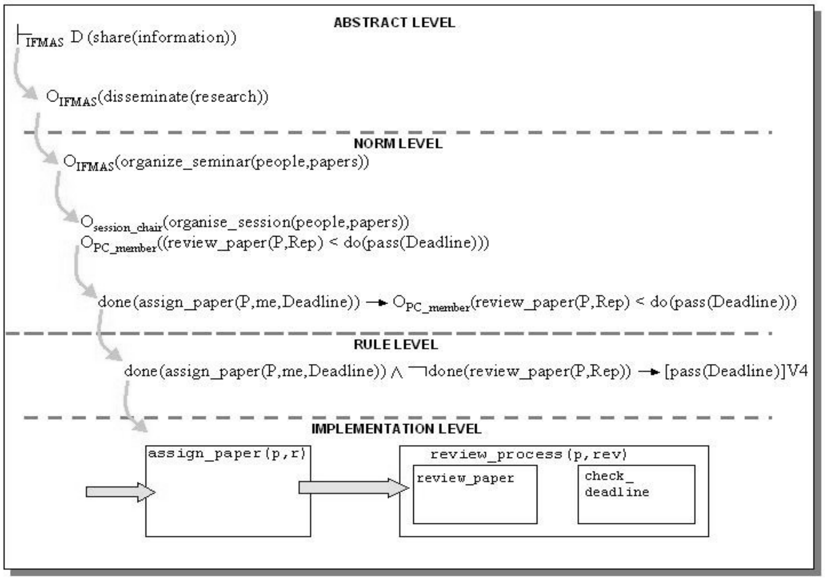

ABSTRACT LEVEL
⊢IFMAS D (share(information))
OIFMAS(disseminate(research))
NORM LEVEL
OIFMAS(organize_seminar(people,papers))
Osession_chair(organise_session(people,papers))
OPC_member((review_paper(P,Rep) < do(pass(Deadline)))
done(assign_paper(P,me,Deadline)) → OPC_member(review_paper(P,Rep) < do(pass(Deadline)))
RULE LEVEL
done(assign_paper(P,me,Deadline)) ∧ ¬done(review_paper(P,Rep)) → [pass(Deadline)]V4
IMPLEMENTATION LEVEL
assign_paper(p,r)
review_process(p,rev)
review_paper
check_deadline

Fig. 5. An example of the refinement process

one might talk of *disseminate*, such action will most probably not be available to agents acting at the implementation level). Specification of communication content is usually realized using ontologies, which are shared conceptualizations of the terms and predicates in a domain. Agent communication languages (ACLs) are the usual means in MAS to describe communicative actions. ACLs are wrapper languages in the sense that they abstract from the content of communication.

In OMNI, the Ontological Dimension describes both the content and the language for communication, at three different levels of abstraction. At the Abstract Level, the *Model Ontology* can be seen as a meta-ontology that defines all the concepts of the framework itself, such as norms, rules, roles, groups, violations, sanctions and landmarks.

The content aspects of communication, or domain knowledge, are specified by *Domain Ontologies*. In OMNI abstract concepts can be iteratively defined and refined in terms of more concrete concepts. The Concrete Domain Ontology includes all the predicates and elements that appear during the design of the Organizational and Normative Structure, and the Procedural Domain Ontology, with the terms from the domain that will be finally used in the implemented system. Concepts or predicates at a lower abstraction level, count as, or implement, concepts at the higher levels. For instance, the actual realization of the AAMAS'04 conference *counts-as* the IFMAS's objective *organize-conference* defined in the Organizational Model, which in turn *counts-as* the IFMAS's value of disseminate knowledge, described in its statutes.

Communication Acts define the language for communication, including the performatives and the protocols. At the Concrete Level, *Generic Communication Acts* define the interactions languages used in the Organizational Model, while the *Specific Communication Acts* covers the communication languages actually used by the agents as they agree in the interaction contracts. As with the content ontologies, communicative acts defined at a lower level of abstraction implement those defined at a higher level.

7 Discussion: Omni Compared with Other Approaches

Development methods for multi-agent systems are currently a hot research topic and several approaches have been proposed. Comprehensive methodologies to design agent societies must be able to describe the characteristics of organizational environments. Such environments are best understood in terms of social concepts such as organization structures, norms and domain language. Furthermore, methodologies must support the development of open societies and the specification of formal institutions. These are issues covered by the Omni framework. In the remainder of this section, we briefly discuss how some well known models support the social and normative concepts introduced by the Omni framework.

GAIA. Gaia [19] is one of the first agent-oriented software engineering methodologies that explicitly takes into account social concepts. Gaia models are situated in at the Concrete Level of Omni (cf. figure 1). While the Implementation Level is explicitly and purposefully ignored, Gaia does not have any notion of an Abstract Level. Gaia is only concerned with the society level and does not capture internal aspects of agent design. However, societies are only considered from the perspective of the individual participants, and therefore Gaia does not deal with communication or other collective issues. Furthermore, normative aspects are reduced to static permissions, a sort of constraints or rules and behavior is fixed in protocols. Moreover, Gaia is not suited to model open domains, and cannot easily deal with self-interested agents, as it does not distinguish between organizational and individual aspects, and does not provide capabilities for agent interpretation of society objectives, norms or plans.

SODA. SODA [11], is actually an extension to Gaia that enables open societies to be designed around suitably-designed coordination media, and social rules to be designed and enforced in terms of coordination rules. As Gaia, SODA distinguishes between an analysis and a design phase. As an attempt to include an higher abstraction level (cf. figure 1), SODA presents a notion of the context, or environment, of the society, albeit not explicit. However, even though SODA distinguishes between agent and collective spaces, it sees roles as the representation of the observable behavior of agents, and therefore cannot represent the difference between the organizational perspective on the activity and aims of individuals (represented by the concept of role in Omni) from the agent perspective on its own activity and aims (represented by the concept of agent in Omni and linked to the role by a social contract). Role enactment is fixed in SODA as the agent model that maps roles to agent classes without any possibility to accommodate agent preferences or characteristics (agent classes are pure specifications of the role characteristics). There are no normative aspects in SODA further than the notion of permission to access infrastructure services. Communication primitives are limited to interaction protocols, and SODA provides no explicit representation for the domain ontology. Furthermore, SODA also does not have a clear and formal semantics.

ISLANDER. The ISLANDER formalism [7] provides a formal framework for institutions [14] and has proven to be well-suited to model practical applications (e.g. electronic auction houses). This formalism views an agent-based institution as a *dialogical system* where all the interactions inside the institution are a composition of multiple dialogic activities (message exchanges). Furthermore, the e-INSTITUTOR platform and the ISLANDER API enable the animation of models and the participation of external agents. The activity of these agents is, however, constrained by *governors* that regulate agent actions, to the precise enactment of the roles specified in the institution model. In contrary to the other frameworks discussed here, ISLANDER provides a sound model for the domain ontology and has a formal semantics [14]. However, ISLANDER provides no primitives to model the Abstract Level of an organization and does not consider the normative aspects of organizations, further than the specification of constraints for scene transition and enactment (the only allowed interactions are those explicitly represented by arcs in scenes).

TROPOS. TROPOS methodology [2] spans the overall development process. It distinguishes between an early and a late requirements phase, and between architectural design and detailed design. The models are implemented using Jack Intelligent Agents [9], which is an agent-oriented extension of Java. Tropos is a fairly complete methodology that considers all steps in system development, and it treats both inter-agent and intra-agent perspectives. The early requirement phase of Tropos, can be seen as a specification of the Abstract Level proposed by OMNI (cf. figure 1). The late requirements phase comes fairly close to the idea of Concrete Level in OMNI , except that it does not provide explicit concepts to capture norms, and ontological aspects are only implicitly described. At the Implementation Level, Tropos provides a detailed implementation of organizational models into JACK agents. The main two shortcomings of Tropos are that a) it is not formal (although there is some ongoing work on providing a formal semantics for Tropos), and b) it is too organizational-centered in the sense that is does not consider that agents can have their own goals and plans, and not just those coming from the organization. Furthermore, Tropos has no concept representing the normative aspects of an organization.

8 Conclusions

In this paper we introduced OMNI, a modelling framework for different types of MAS, from closed systems to open, flexible environments. This approach is rather unique, as most existing frameworks concentrate on a specific type of MAS.

The modular structure of OMNI facilitates the adaptation of the framework to different types of domains. In those domains with none or small normative components, design is guided by the Organizational Dimension, while in highly regulated domains the Normative Dimension is more prominent and therefore guides the design.

All dimensions of OMNI have a formal logical semantics, which ensures consistency and possibility of verification of the different aspects of the system. For more information on the formalization aspects, we refer the reader to [5] for a detailed specification of the formalization of the organizational and ontological dimensions, and to [17] for the formal normative model.

Currently we are taking the first steps towards implementing tools to use with the framework. We will be using ISLANDER as a basis for the support of the implementation level and build the other levels on top of that.

References

1. G. Abdelkader. Requirements for achieving software agents autonomy and defining their responsibility. In *Proc. Autonomy Workshop at AAMAS 2003.*
2. P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. TROPOS: An agent-oriented software development methodology. *Journal of Autonomous Agents and Multi-Agent Systems*, to appear.
3. M. Dastani, V. Dignum, and F. Dignum. Role assignment in open agent societies. In *Proc. AAMAS'03*, 2003.
4. F. Dignum. Autonomous Agents with Norms. *AI and Law*, 7:69–79, 1999.
5. V. Dignum. *A Model for Organizational Interaction: based on Agents, founded in Logic.* SIKS Dissertation Series 2004-1. SIKS, 2004. PhD Thesis.
6. V. Dignum and F. Dignum. Modeling agent societies: Coordination frameworks and institutions. In P. Brazdil and A. Jorge, editors, *Progress in Artificial Intelligence*, LNAI 2258, pages 191–204. Springer-Verlag, 2001.
7. M. Esteva, J. Padget, and C. Sierra. Formalizing a language for institutions and norms. In J.-J.CH. Meyer and M. Tambe, editors, *Intelligent Agents VIII*, volume 2333 of *LNAI*, pages 348–366. Springer Verlag, 2001.
8. D. Grossi and F. Dignum. Abstract and concrete norms in institutions. *Accepted in FAABS 2004*, 2004.
9. Nick Howden, Ralph Rönquist, Andrew Hodgson, and Andrew Lucas. Jack - summary of an agent infrastructure. In *5th International Conference on Autonomous Agents*. 2001.
10. J.-J. Ch. Meyer. A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic. *Notre Dame J. of Formal Logic*, 29(1):109–136, 1988.
11. A. Omicini. Soda: Societies and infrastructures in the analysis and design of agent-based systems. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering*, volume 1957 of *LNAI*, pages 185–193. Springer Verlag, 2001.
12. A. Omicini. Towards a notion of agent coordination context. In D. Marinescu and C. Lee, editors, *Process Coordination and Ubiquitous Computing*, pages 187–200. CRC-Press, 2002.
13. H.V.D. Parunak and J. Odell. Representing social structures in uml. In M.Wooldridge, G.Weiss, and P. Ciancarini, editors, *Agent-Oriented Software Engineering II*, LNCS 2222. Springer-Verlag, 2002.

14. J.A. Rodriguez. *On the Design and Construction of Agent-mediated Electronic Institutions*. PhD thesis, Institut d'Investigació en Intel.ligència Artificial (IIIA), 2001.
15. I. Smith, P. Cohen, J. Bradshaw, M. Greaves, and H. Holmback. Designing conversation policies using joint intention theory. In *Proc. ICMAS-98*, pages 269–276. IEEE Press, 1998.
16. W. Vasconcelos, J. Sabater, C. Sierra, and J. Querol. Skeleton-based agent development for electronic institutions. In *Proc. AAMAS'02*, 2003.
17. J. Vázquez-Salceda. *The Role of Norms and Electronic Institutions in Multi-Agent Systems*. Whitestein Series in Software Agent Technology. Birkhauser Verlag AG, Switzerland, 2004.
18. J. Vázquez-Salceda and F. Dignum. Modelling electronic organizations. In V. Marik, J. Muller, and M. Pechoucek, editors, *Multi-Agent Systems and Applications III*, LNAI 2691, pages 584–593. Springer-Verlag, 2003.
19. M. Wooldridge, N.R. Jennings, and D. Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.
20. F. Zambonelli. Abstractions and infrastructures for the design and development of mobile agent organizations. In M.Wooldridge, G.Weiss, and P. Ciancarini, editors, *Agent-Oriented Software Engineering II*, LNCS 2222, pages 245–262. Springer-Verlag, 2002.
21. F. Zambonelli, N. Jennings, and M. Wooldridge. Organisational abstractions for the analysis and design of multi-agent systems. In P. Ciancarini and M.Wooldridge, editors, *Agent-Oriented Software Engineering*, LNCS 1957, pages 98–114. Springer-Verlag, 2001.

A Dialogue Game to Offer an Agreement to Disagree

Henk-Jan Lebbink[1,2,3], Cilia Witteman[2], and John-Jules Ch. Meyer[1]

[1] Institute of Information and Computing Sciences,
Utrecht University, The Netherlands
henkjan@cs.uu.nl
[2] Social Sciences, Radboud University Nijmegen, The Netherlands
[3] Emotional Brain BV, Almere, The Netherlands

Abstract This paper proposes a dialogue game in which coherent conversational sequences at the speech act level are described of agents that become aware they have an irresolvable disagreement and settle the dispute by agreeing to disagree. A disagreement is irresolvable from an agent's perspective if the agent is aware that both parties have ran out of options to resolve the dispute, and that both parties are aware of this. A dialogue game is formulated in which agents can offer information that may unintentionally result in irreconcilable, mutually inconsistent belief states. Based on the agents' cognitive states, dialogue rules and cognitive rules are defined that allow agents to come to an agreement to disagree. These rules are implemented in the programming language Prolog, resulting in an intuitive design for multi-agent systems.

1 Introduction

If our goal is to understand human conversations, we may need to model carefully their underlying principles, but for generating communication in multi-agent systems, we may be satisfied if we can build computational models that generate efficient and useful conversations. In conversations in general, participating agents have autonomy over their cognitive states, but they may also have desires to change those of others. In trying to do so, these agents may find themselves stuck in impasses over irreconcilable beliefs. This paper addresses the questions how to cope with these impasses and how to devise a computational model to identify irreconcilable beliefs from an agent's local perspective. We will use dialogue games to define reasoning and communication rules to overcome such situations.

Dialogue games have recently received more attention in the field of computer science, and, especially, in the community of multi-agent systems [1, 2]. In multi-agent systems, autonomous software agents communicate and cooperate to reach private and collective goals. We will not address issues related to cooperation, but we will focus primarily on agents engaging in communication. A dialogue typology by Walton and Krabbe [3] identifies four different categories of dialogues by distinguishing the agents' initial situations and goals. The categories are:

Persuasion dialogues, in which agents seeks to convince other agents to believe propositions [4, 5, 6]; Negotiation dialogues, in which participants seek to agree on how to divide a resource [7, 8]; Deliberation dialogues, in which participants make plans by discussing which actions to perform in which situations [9] or which beliefs to accept to minimize uncertainty [10] and which result in collective intentions [11] or group-plans and teams [5]; And information seeking dialogues, in which agents seeks to find truth-values of propositions by asking others who may have answers [12, 13, 14]. The current work contributes to the category of persuasion dialogues, and, especially, when persuasion dialogues terminate.

Beun [13] and Lebbink et al. [14] describe communicative acts and communication rules that form dialogue games that agents play to balance their desires and belief states. Such a dialogue game consists of pre-conditions for uttering communicative acts to convey information to other agents, and post-conditions that state the agents' cognitive states after incoming and outgoing information is processed. To describe inconsistent and biased information states, a multi-valued logic [14] is used with which agent can have inconsistent belief states without being forced to perform belief revision. We formulate a semantics for communicative acts to offer information to agree to disagree in the same vein as the FIPA work on agent communication languages [15, 16].

What is lacking in Beun [13] and Lebbink et al. [14] is the possibility for agents to recognize irresolvable disagreements and, based on this recognition, to utter an agreement to disagree, thereby making the disagreement common belief. This common belief may motivate dialogues to redefine the meaning of formulae that resulted in the disagreement.

Agents may be motivated to persuade others to decide to believe certain propositions, for example, when agents participate in group-plans that require cooperation; they may need to agree on certain propositions for the plan to succeed. Consequently, agents may need to offer propositions to others, and, in response, agree to believe propositions, or, on the other hand, reject to believe these propositions when accepting them would result in inconsistent belief states. Our objective is to present a dialogue game in which cognitive agents become aware of irreconcilable beliefs and show this awareness to others while preserving their private beliefs.

In Section 2, our agent architecture is presented in which agents can have inconsistent beliefs and desires; in addition, a reasoning game is defined allowing agents to decide to believe propositions. A dialogue game is presented in Section 3 enabling agents to offer information. In Section 4, the reasoning game is extended to allow agents to agree to disagree. The resulting formalism permits embedded dialogues, verification of existing dialogues, and a straightforward implementation due to its computational nature (Section 5).

2 Agent Architecture

2.1 Dialogue Games

Early approaches to the semantics of propositions in the philosophy of language centred on the view that semantics are truth-conditional. Searle [17] and Austin

[18] –and many others dating back to Aristotle– adhere to this view, explicitly embracing the idea that truth consists in a relation to reality: a belief is true when there is a corresponding fact, and false when there is no corresponding fact. This is what is called the 'correspondence theory of truth': the truth of a proposition is inseparable from reality [19].

However, demanding strict correspondence between truth and reality makes it rather difficult –if not impossible– for agents to have true beliefs, for then they need to know some segment of reality. A practical and obvious problem with this semantics is that some propositions have an inherent uncertainty: most past and future propositions are uncertain; not to mention the epistemological problem how an agent is to *know* what reality really is. (See [20] on the question "do you believe in reality?")

The correspondence theory has been criticised by Dummett [21] and Wright [22], among others, who advocate a verificationist semantics, lifting the burden for agents from knowing reality to verifying evidence. Agents can derive knowledge from a process of inquiry in which a chain of mental and physical intermediaries connect. However, agents can never be sure whether propositions can be treated as knowledge or as possible false beliefs that need further investigation. The underlying semantics of the correspondence theory, proposed by mathematicians Frege and Tarski [23], has also been criticised by the later Wittgenstein. However, Ellenbogen [24] shows that Dummett's notion that certain propositions which we treat as uncertain also rests upon a realist conception of truth and that "his argument ultimately rests on a refusal to recognize an alternative account of what it is to determine the truth value of a sentence..." [24–p.26]. We will not use the verificationist semantics because it presumes that agents have absolute knowledge of the world; what we will use is a semantics based on "use". In Philosophical Investigations [25], Wittgenstein proposes "use" as an alternative to construct the semantics of propositions.

> *"According to the dictum "meaning is use", what makes it correct to call a statement "true" is not its correspondence with how things are, but our criterion for determining its truth. What it means for us to call a statement "true" is that we currently judge it true, knowing that we may some day revise the criteria whereby we do so."* [24–cover page]

Wittgenstein proposes "language games" with several different however related uses. Our focus is not on how language games can be learned, nor how they refer to a multiplicity of language practices in our ordinary languages. We use its reference to models of primitive language that Wittgenstein has invented to clarify the working of language in general. In his view, communicative acts only have meaning within a particular language game: acts outside a game are just meaningless and useless syntactic structures. Language games provide rules of usage: they define when agents are allowed to pose and answer communicative acts; and, additionally, language games provide rules how the agent's cognitive state changes due to communication. Instead of presupposing that agents need to know reality or other agents' cognitive states to verify factual statements, language games are formulated that describe inquiry with reality or agents. In such inquiry games, currently, results obtained from, for example, microscopes or lie detectors are considered truth-bearers; results obtained from witchcraft are not. However, this may change as science progresses.

We propose two language games, one game that defines the meaning of communicative acts to handle information offers like in contemporary agent communication languages [15, 16], and one game that defines the meaning of decisions to believe propositions [26]. The former will be called a *dialogue game*, the latter a *reasoning game*. Remember that meaning set forward in games is understood as rules of usage. The games provide sets of pre-conditions that define which communicative acts or decisions agents may perform given their current cognitive states. In addition, games provide sets of post-conditions that define the contents of the agent's cognitive state after communicative acts are uttered or decision are made. Pre and post-conditions are combined to form dialogue or reasoning rules.

We assume that information can only accumulate in the participants' cognitive states, and cannot be retracted. In these information-monotonic games, additions may introduce inconsistent beliefs. The reasoning game for deciding to believe propositions stipulates what it means to have inconsistent beliefs: the agent is aware of at least two sources that have sufficient and equal persuasive powers, thereby rendering her belief state inconsistent. To have an inconsistent belief state does not mean there is a segment of reality that corresponds to this belief, but that an agent is convinced by two equally persuasive sources. Although we present a reasoning game that prevents agents from deciding to believe propositions that would result in inconsistent belief states, agents use the possibility of future inconsistencies in a look-ahead fashion in their decisions to believe proposition.

Agents can only speak to one agent at a time via an ideal half-duplex communication channel, which means that no information is lost and that information can only flow in one direction at a time. No restrictions are made on the number of participants in the dialogue, agents are assumed to be omniscient, and aware they use the same dialogue and reasoning games.

2.2 Example Dialogue

Consider the following fictitious dialogue between two Sesame Street puppets. Tv wants to insure his Ferrari. To achieve this, he rings an insurance company and explains the situation by stating his desire to an insurance agent (Ia for short). The Ia wants to sell Tv an expensive insurance policy, because Sesame Street puppets are notoriously prone to fast and dangerous driving, especially in Ferraris. What the Ia wants is that Tv accepts that his car is not safe, which justifies an expensive policy. The dialogue consists of offerings of propositions that are either accepted or declined. For the sake of argument, both agents are rather stubborn and will stick to their first beliefs, and do not accept to believe information that would render their belief state inconsistent.

Example 1 (Dialogue about car insurance in Sesame Street).

1.	Tv to Ia	‘My car is a Ferrari.’
2.	Ia to Tv	‘Ok.’
3.	Tv to Ia	‘(and) My car is safe.’
4.	Ia to Tv	‘I don’t believe that.’
5.	Ia to Tv	‘(actually) I think your car is not safe.’
6.	Tv to Ia	‘I don’t accept that my car is not safe.’
7.	Ia to Tv	‘Do you accept that if a car is a Ferrari then it’s not a safe car?’
8.	Tv to Ia	‘(no) I don’t want to accept that.’
9.	Ia to Tv	‘Lets agree to disagree whether your car is safe or not.’
10.	Tv to Ia	‘Ok.’

In the remainder of this paper, the example dialogue will be shown to be a valid sequence in our dialogue game starting from initial cognitive states (from Example 3). To achieve this result, the example dialogue is translated to a sequence of communicative acts (Section 3) with propositions taken from an ontology (from Example 2).

2.3 Multi-valued Logics

Whereas in classical logic propositions are assigned truth-values *true* or *false*, later, in philosophy ‘deviant’ logics were developed that are capable of representing uncertain, non-determined states or epistemic attitudes [27]. More recently, other truth-qualities were required by computer scientists for efficient software implementations, for example, in the verification of circuit board design [28]. In the field of multi-agent systems, computer scientists use non-classical truth-values and semantics, predominantly modal logics, to represent the agents’ cognitive states [29]. Our approach is to model the agent’s cognitive state with multi-valued logics (MVL). In these logics, propositions are primitive formulae that are assigned multiple truth-values from a bilattice structure [30, 31].

Bilattice Structures. Different modalities are needed to represent the agent’s cognitive state. In the MVL introduced in Lebbink et al. [14], propositions are constructed in a fashion that is considered truth-value bearing, capable of being the object of belief or ignorance. This MVL can represent a lack of information (*unknown*) as well as over-informative states (*inconsistent*); the truth-values are taken from a bilattice structure.

A bilattice is an algebraic structure that formalises a space of generalized truth-values with two lattice orderings [30, 31]. The bilattice for a four-valued logic, proposed by Belnap [32], is graphically depicted in Figure 1. Truth-values t and f stand for the classical truth-values *true* and *false* respectively; non-orthodox truth-values u and i represent a complete lack of information (*unknown*) and the inconsistent information state (*inconsistent*). Truth-values are ordered by the amount of truth $\leq_t$ and the amount of information $\leq_k$; currently, only the latter order is of interest. For instance, *unknown* has less information than *true* and *false*, denoted by $\mathsf{u} \leq_k \mathsf{t}$ and $\mathsf{u} \leq_k \mathsf{f}$. Truth-values *true* and *false* are

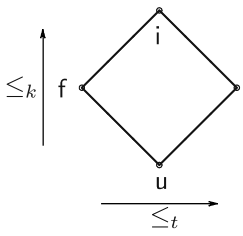


Fig. 1. Smallest complete bilattice for a four-valued logic proposed by Belnap

unrelated to one another in the k-order, that is, $\mathsf{t} \not\leq_k \mathsf{f}$ and $\mathsf{f} \not\leq_k \mathsf{t}$. Bilattices with more truth-values and even a continuum of truth-values can be used to represent biased information or probabilities [30]; we use only the truth-values from Figure 1.

The greatest k-lower bound $\otimes_k$ can be thought of as the truth-value representing the information that is shared by the two truth-values, that is, the mutual information of the two truth-values, for example, $\mathsf{f} \otimes_k \mathsf{t} = \mathsf{u}$. Likewise, the least k-upper bound $\oplus_k$ is thought of as the information that results after combining the two truth-values, for example, $\mathsf{f} \oplus_k \mathsf{t} = \mathsf{i}$. See Ginsberg [30] and Fitting [31] for a formal treatment of the bilattice operators.

Language of MVL. In MVL, atomic and non-atomic MVL propositions are distinguished. The atomic MVL proposition $p{:}\theta$ is a formula p taken from an ontology $\mathcal{O}$ with a truth-value θ from a bilattice structure $\mathcal{B}$. The proposition $p{:}\theta$ is read as "formula p has at least truth-value θ". The formula is said to have at least the information represented in the truth-value. In the sequel, we will speak of propositions instead of MVL propositions. For our current purposes, an ontology is a set of primitive formulae; for ontologies with more structure, see for example Sowa [33].

The non-atomic proposition $(\psi \rightarrowtail \varphi){:}\theta$ is an inference rule with truth-value θ. Proposition ψ is the antecedent for proposition φ; φ is the consequent of the inference rule. In case θ equals t, the inference rule is written as $\psi \rightarrowtail \varphi$. Remark that $\rightarrowtail$ is not a normal connective but a formula, part of the logical language that codes an inference rule in the object language. Also remark that the truth-values of antecedents and consequents are embedded in the rule. This nesting of sub-sentences is non-standard in MVL, but blurring syntax and semantics will not introduce problems. Other connectives are not defined.

A special purpose proposition $a2d(x, y, \psi, \varphi) : \theta$ states that agent x and y, member of a set of agents $\mathcal{A}$, agree to disagree on propositions ψ and φ, this is further discussed in Section 4. The set of truth-values is denoted C.

Definition 1 (Language of MVL). *Given bilattice $\mathcal{B} = \langle C, \leq_k, \leq_t \rangle$, ontology $\mathcal{O}$, and agents $\mathcal{A}$, the* language of MVL $\mathcal{L}^{\mathcal{B}}$ *is the smallest set satisfying:*

1. *if $p \in \mathcal{O}$ and $\theta \in C$ then $p{:}\theta \in \mathcal{L}^{\mathcal{B}}$,*
2. *if $\psi, \varphi \in \mathcal{L}^{\mathcal{B}}$ and $\theta \in C$ then $(\psi \rightarrowtail \varphi){:}\theta \in \mathcal{L}^{\mathcal{B}}$,*
3. *if $\psi, \varphi \in \mathcal{L}^{\mathcal{B}}$, $x, y \in \mathcal{A}$ and $\theta \in C$ then $a2d(x, y, \psi, \varphi){:}\theta \in \mathcal{L}^{\mathcal{B}}$.*

Theories of MVL. Five deduction rules are defined which will be used to construct theories of MVL. Instead of theories of MVL, we will speak of theories.

The complete lack of information associated with truth-value *unknown* always applies to all propositions present in a theory. Remember that $p{:}\theta$ reads that p has at least truth-value θ; therefore, all propositions of the language of MVL have a minimal and unique information state *unknown* in a theory.

$$p{:}\theta \in \mathcal{L}^{\mathcal{B}} \quad \Longrightarrow \quad p{:}\mathsf{u} \in \mathcal{T} \tag{R1}$$

The reading of propositions enforces that if a proposition is part of a theory, then all propositions with less information are also part of the theory. If proposition $p{:}\theta_1$ has at least truth-value θ_1, and θ_2 represents less information than θ_1, then formula p also has at least truth-value θ_2. The information in $p{:}\theta_2$ is said to be subsumed under $p{:}\theta_1$.

$$p{:}\theta_1 \in \mathcal{T} \quad \& \quad \theta_2 \leq_k \theta_1 \quad \Longrightarrow \quad p{:}\theta_2 \in \mathcal{T} \tag{R2}$$

Information is closed in a theory if the least k-upper bound of truth-values of the same formula present in the theory is also present. Remember that the least k-upper bound is thought of as the information that results from combining two truth-values. For example, interpret the theory as an agent's belief state. If the agent believes that p is *true*, and, at the same time, that p is *false*, then the agent also believes that p is *inconsistent*.

$$p{:}\theta_1 \in \mathcal{T} \quad \& \quad p{:}\theta_2 \in \mathcal{T} \quad \Longrightarrow \quad p{:}\theta_1 \oplus_k \theta_2 \in \mathcal{T} \tag{R3}$$

Dual theories are theories with an ordering $\leq_k^\delta$ with $\theta_1 \leq_k^\delta \theta_2 = \theta_2 \leq_k \theta_1$. Due to this reversed order, the least and unique information state from R1 is reversed. All propositions part of the language of MVL have a unique minimal state i in a dual theory.

$$p{:}\theta \in \mathcal{L}^{\mathcal{B}} \quad \Longrightarrow \quad p{:}\mathsf{i} \in \mathcal{T} \tag{R1d}$$

The reading of propositions also enforces subsumed information in dual theories. If a proposition is part of a dual theory, then all propositions with more information are also present in the dual theory.

$$p{:}\theta_1 \in \mathcal{T} \quad \& \quad \theta_2 \leq_k^\delta \theta_1 \quad \Longrightarrow \quad p{:}\theta_2 \in \mathcal{T} \tag{R2d}$$

Theories of MVL are defined as sets of propositions closed under deduction rules. We denote by $Cn^d(\Psi, \mathcal{R})$ the set of propositions that results from $\Psi \subseteq \mathcal{L}^{\mathcal{B}}$ closed under the set of deduction rules $\mathcal{R}$. If ambiguity is unlikely to occur we write $Cn^d(\Psi)$ instead of $Cn^d(\Psi, \mathcal{R})$.

Definition 2 (MVL Theory). *Given a language of MVL $\mathcal{L}^{\mathcal{B}}$, three MVL theories, $\mathcal{T}, \mathcal{T}^c, \mathcal{T}^\partial \subseteq \mathcal{L}^{\mathcal{B}}$ are defined by the closure under deduction rules.*

- *(Normal) theory $\mathcal{T} = Cn^d(\mathcal{T}, \{R1, R2\})$;*
- *Complete theory $\mathcal{T}^c = Cn^d(\mathcal{T}^c, \{R1, R2, R3\})$;*
- *Dual theory $\mathcal{T}^\partial = Cn^d(\mathcal{T}^\partial, \{R1d, R2d\})$.*

Interpretation. An example of a closed theory is an agent's belief state. An agent has at least no information about a proposition, that is, it is not possible to be less informed about a formula p than p:u (Rule R1). If an agent believes that formula p has at least truth-value *inconsistent* then she also believes that p has at least truth-value *true* and *false* (Rule R2). If an agent believes a proposition p:t and she concludes to believe p:f, then she also believes p:i which means that she has an inconsistent belief state with regard to formula p (Rule R3).

An example of a dual theory is that of an agent's ignorance state. Consider the situation in which an agent x keeps record of another agent y's ignorance state. If agent x believes that y is ignorant about p:t, that is, x believes that y does not believe p:t, then x also believes that y is ignorant about p:i (Rule R2d).

2.4 The Agent's Cognitive State

An agent's cognitive state consists of a finite number of mental states, which are theories of MVL. We will not present a full repertoire of all possible mental states agents have regarding themselves and others; only those are identified that are used in the present paper. Remember that set $\mathcal{A}$ denotes the set of agent identifiers.

- Private belief B_x is a complete theory denoting agent x's beliefs. For instance, p:t $\in B_x$ states that x believes that formula p has at least truth-value *true*.
- Private desire to believe D_xB_y is normal theory with $y \in \mathcal{A}$ and *not* $x \neq y$, denoting agent x's desires that agent y is to believe. $\psi \in D_xB_y$ states that agent x desires that agent y is to believe ψ, and $\psi \in D_xB_x$ denotes that x desires to believe ψ.
- Manifested belief B_xB_y is a complete theory with $y \in \mathcal{A}$ and $x \neq y$, denoting the beliefs of y that x keeps record of. For instance, $\psi \in B_xB_y$ states that x is aware that y believes ψ.
- Manifested desires to believe $B_xD_yB_x$ is a normal theory with $y \in \mathcal{A}$ and $x \neq y$, denoting x's awareness of y's desire that x is to believe.
- Manifested ignorance state B_xI_y is a dual theory with $y \in \mathcal{A}$ and $x \neq y$, denoting the propositions that y does not believe that x is aware of. $\psi \in B_xI_y$ states that agent x is aware that agent y is ignorant of ψ.
- In addition, other higher-order manifested mental states are defined likewise. $B_xB_yB_x$ is a complete theory, $B_xB_yI_x$, $B_xI_yB_x$ and $B_xB_yI_xB_y$ are dual theories, $B_xB_yD_xB_x$ and $B_xB_yB_xD_yB_x$ are theories; other mental states like $B_xI_yI_x$ and $B_xI_yD_xB_x$ are not discussed and not used.

The above-mentioned mental states are part of a structure CS which represents the agent's cognitive state: we mean by $CS_x \models \Pi$ the set of set-theoretical propositions Π that hold for agent x's cognitive state. For example, to state that agent x does not believe that agent y believes ψ, and, at the same time does desire that agent y believes ψ, is denoted $CS_x \models \{(\psi \notin B_xB_y),\ (\psi \in D_xB_y)\}$. If Π is a singleton set, it is substituted with its element; for example, we write $CS_x \models \psi \in B_x$ instead of $CS_x \models \{\psi \in B_x\}$. We write $\{CS_x, CS_y\} \models \Pi$ instead of $\forall \pi \in \Pi\ (CS_x \models \pi$ 'or' $CS_y \models \pi)$.

Example 2 (Ontology). To model the dialogue from Example 1, two primitive formulae are needed: $\mathsf{is\_a}(\mathsf{this\_car}, \mathsf{ferrari})$ and $\mathsf{is\_a}(\mathsf{this\_car}, \mathsf{safe})$ which state that the object $\mathsf{this\_car}$ is an instance of the class $\mathsf{ferrari}$, and that it is of class safe, we have: $\mathsf{is\_a}(\mathsf{this\_car}, \mathsf{safe}),\ \mathsf{is\_a}(\mathsf{this\_car}, \mathsf{ferrari}) \in \mathcal{O}$.

The cognitive state of all participating agents will be called a collective cognitive state.

Example 3 (Initial collective cognitive state). By default, Tv believes that his car is a Ferrari and a safe one. Next to that, Tv desires that the Ia believes that its car is a Ferrari and safe. On the other hand, the Ia desires by default, that Tv believes its car is not safe; and, in addition, the Ia believes that if a car is a Ferrari then the car is not safe. The agents do not have other desires or beliefs. The superscript 1 denotes the initial cognitive state, consecutive numbers denote cognitive states as the dialogue unfolds. Together with the ontology from Example 2 we have:

$$\begin{aligned}
CS^1_{\mathsf{ia}} \models \{\ & \mathsf{is\_a}(\mathsf{this\_car}, \mathsf{safe})\mathsf{:f} \in D_{\mathsf{ia}} B_{\mathsf{tv}}, \\
& \mathsf{is\_a}(\mathsf{this\_car}, \mathsf{ferrari})\mathsf{:t} \rightarrowtail \mathsf{is\_a}(\mathsf{this\_car}, \mathsf{safe})\mathsf{:f} \in B_{\mathsf{ia}}\ \} \\
CS^1_{\mathsf{tv}} \models \{\ & \mathsf{is\_a}(\mathsf{this\_car}, \mathsf{ferrari})\mathsf{:t} \in D_{\mathsf{tv}} B_{\mathsf{ia}}, \\
& \mathsf{is\_a}(\mathsf{this\_car}, \mathsf{ferrari})\mathsf{:t} \in B_{\mathsf{tv}}, \\
& \mathsf{is\_a}(\mathsf{this\_car}, \mathsf{safe})\mathsf{:t} \in D_{\mathsf{tv}} B_{\mathsf{ia}}, \\
& \mathsf{is\_a}(\mathsf{this\_car}, \mathsf{safe})\mathsf{:t} \in B_{\mathsf{tv}}\ \}
\end{aligned}$$

2.5 A Reasoning Game to Decide to Believe Propositions

Different definitions when agents are to decide to believe propositions are possible: one could state that agents are allowed to decide to believe a proposition if they themselves believe the criteria to deduce that proposition with an inference rule. We add to this capability the situation in which agents conform to other agents' beliefs.

Cognitive processes are prescribed with reasoning rules that define when agents are allowed to make decisions, and the effects these decisions have on the agents' cognitive state. This is done by specifying sets of pre and post-conditions. Currently, only decisions to add propositions to belief and desire states are possible. Three cognitive processes for making a decision to believe proposition are distinguished: (1) deducing consequences of private beliefs with inference rules; (2) deciding to believe propositions based on other agents' beliefs; and, (3) deducing that one has an irresolvable disagreement with another agent. The latter cognitive process is described in Section 4.

Reasoning Game. A reasoning game is a finite set of reasoning rules that allow agents to make decisions according to the pre and post-conditions of specific decisions. A decision's pre and post-conditions are combined in a reasoning rule, providing the semantics of the decision in a "meaning is use" fashion.

A generic reasoning rule for an abstract decision $\lambda(x, \psi, ms)$ is defined, which can be instantiated with concrete reasoning rules. The decision $\lambda(x, \psi, ms)$ regarding some proposition ψ and mental state ms is allowed for agent x if the set of pre-conditions of $\lambda(x, \psi, ms)$ holds in her cognitive state, that is, $CS_x \models pre(\lambda(x, \psi, ms))$. After the decision is made, the agent's cognitive state is updated, resulting in a new cognitive state in which the post-condition of $\lambda(x, \psi, ms)$ holds, that is, $CS'_x \models post(\lambda(x, \psi, ms))$. Note that the post-conditions never require propositions *not* to be part of mental states; this holds for both reasoning rules and dialogue rules (Section 3.2) both resulting in information monotonic theory updates.

$$CS_x \models pre(\lambda(x, \psi, ms)) \quad \Longrightarrow \quad CS'_x \models post(\lambda(x, \psi, ms)) \qquad \text{(RR)}$$

We are interested in cognitive states closed under sets of reasoning rules. By $Cn^r(CS_x, \mathcal{R})$ we denote agent x's cognitive state that results from the closure under the set of reasoning rules $\mathcal{R}$. If ambiguity is unlikely to occur, we write $Cn^r(CS_x)$ instead of $Cn^r(CS_x, \mathcal{R})$, and if $\mathcal{R}$ is a singleton set, it is substituted with its element. If the sets of pre and post-conditions are confined to propositions with regard to one mental state, for example, the agent x's belief state, one may want to write $Cn^r(B_x, \mathcal{R})$ instead of $Cn^r(CS_x, \mathcal{R})$. Note that the set $Cn^r(\mathcal{T}, \mathcal{R})$ yields a theory instead of an entire cognitive state.

Deducing Consequences. Agents may deduce new beliefs that are based on their current beliefs. If an agent holds the belief that an inference rule linking two propositions has a designated truth-value *true*, and she believes the antecedent of the inference rule, then the agent may deduce the consequent and add this inferred proposition to her belief state.

If agent x believes ψ and inference rule $\psi \rightarrowtail \varphi$, then she may deduce φ. That is, x may decide to believe φ. These two pre-conditions are part of the reasoning action $d2a_1(x, \varphi, B_x)$ that denotes that x decides to add φ to its mental state B_x. With Reasoning Rule D2A$_1$ we mean the generic Reasoning Rule RR instantiated for this decision to believe a proposition. Different decisions are indexed to distinguish different sets of pre-conditions.

$$(\psi \in B_x),\ (\psi \rightarrowtail \varphi \in B_x) \ \in \ pre(d2a_1(x, \varphi, B_x))$$

Note that only inference rules with a truth-value *true* are used and that the blurring of syntax and semantics due to nesting of sub-formulas is minimal. Inference rules with truth-value $\theta \not\geq_k \mathsf{t}$, for example *false*, do not have a straightforward interpretation; *true* states that there is a relation between the consequent and the antecedent, *unknown* states that there is no relation. However, what *false* could denote is not clear. (See Rescher [27] on the notion of designated truth-values and consequence relations in MVLs.)

If the pre-conditions of the decision $d2a_1(x, \psi, B_x)$ hold, the agent is allowed to perform the act of deciding to believe ψ, resulting in the state in which ψ is contained in x's belief state. The set of post-conditions is straightforward.

$$(\psi \in B_x) \ \in \ post(d2a(x, \psi, B_x))$$

Example 4 (Ia decides to believe). Abbreviate ψ for $\mathsf{is\_a(this\_car, ferrari){:}t}$, φ for $\mathsf{is\_a(this\_car, safe){:}f}$ and assume $CS^2_{\mathsf{ia}} \models \psi \rightarrowtail \varphi \in B_{\mathsf{ia}}$. If the Ia is persuaded to believe ψ, then with Reasoning Rule D2A$_1$ the Ia decides to believe φ. However, if $CS^2_{\mathsf{ia}} \models \mathsf{is\_a(this\_car, safe){:}t} \in B_{\mathsf{ia}}$ already holds, then so does $CS^{2'}_{\mathsf{ia}} \models \mathsf{is\_a(this\_car, safe){:}i} \in B_{\mathsf{ia}}$ with Deduction Rule R3.

Next to agents' deducing new beliefs for themselves, agents may also deduce that other agents deduce new beliefs. An agent x may deduce that y should decide to add proposition ψ to her belief state, if x believes that y believes φ and $\varphi \rightarrowtail \psi$. This decision is denoted $d2a_1(x, \psi, B_x B_y)$, which is read as "agent x decides to add ψ to its belief about agent y's beliefs".

$$(\varphi \in B_x B_y),\ (\varphi \rightarrowtail \psi \in B_x B_y) \ \in \ pre(d2a_1(x, \psi, B_x B_y))$$

Agents may even deduce what other agents can deduce about their beliefs.

$$(\varphi \in B_x B_y B_x),\ (\varphi \rightarrowtail \psi \in B_x B_y B_x) \ \in \ pre(d2a_1(x, \psi, B_x B_y B_x))$$

Conformism. The second possibility for an agent to obtain new beliefs is by conforming to other agents' belief states. If an agent believes that another agent believes a proposition, and the agent does not herself believe the proposition, then she may decide to believe the proposition if the other agent is trustworthy. It must be noted that agents have no choice but to decide to believe a proposition: if the pre-conditions of the decision are met, then the agent needs to act accordingly, in effect deciding to perform the decision. The abstract Reasoning Rule RR is instantiated to form the rule for decision $d2a_2(x, \psi, ms)$, this rule is denoted D2A$_2$.

Agent x may decide to believe proposition ψ if x is aware that another agent y believes ψ and x does not already believe ψ. Agent x's decision to believe ψ in conformity to another agent's belief state is denoted by $d2a_2(x, \psi, B_x)$.

$$(\psi \in B_x B_y),\ (\psi \notin B_x) \ \in \ pre(d2a_2(x, \psi, B_x))$$

An additional pre-condition to conforming to another agent's belief state is that agents may only decide to believe propositions if these do not introduce new inconsistencies. Stated differently, for every inconsistent proposition present in a belief state after addition of ψ, holds that this inconsistent proposition was already present before ψ was added.

$$\big(\forall p{:}\mathsf{i} \in \mathcal{L}^{\mathcal{B}}\ (p{:}\mathsf{i} \in Cn^r(B_x \cup \{\psi\}, \mathrm{D2A}_1) \Rightarrow p{:}\mathsf{i} \in B_x)\big) \ \in \ pre(d2a_2(x, \psi, B_x))$$

As a result of the previous pre-condition, agents decide to believe propositions in a first-come-first-serve basis, making the order of uttering communicative acts of importance for the outcome of the dialogue.

Example 5 (Ia decides to believe). Abbreviate ψ for $\mathsf{is\_a(this\_car, ferrari){:}t}$. The Ia may perform a $d2a_1(\mathsf{ia}, \psi, B_x)$ if she does not believe ψ; however, she does believe that Tv believes ψ, $CS^2_{\mathsf{ia}} \models \{\psi \notin B_{\mathsf{ia}},\ \psi \in B_{\mathsf{ia}} B_{\mathsf{tv}}\}$. In this cognitive state, Reasoning Rule D2A$_2$ is applicable resulting in state $CS^{2'}_{\mathsf{ia}} \models \psi \in B_{\mathsf{ia}}$.

3 A Dialogue Game to Offer Information

3.1 Communicative Acts

The dialogue game to offer information provides semantics for three syntactically different communicative acts. A communicative act $\lambda(x, y, \psi)$ is uttered by speaker x directed to addressee y regarding proposition ψ.

With the communicative act $oba(x, y, \psi)$ the addressee y is offered a proposition ψ with the request to decide to believe it, the act is read as “Are you (y) willing to decide to believe proposition ψ?”. The abbreviation oba is short for offer a belief addition. The communicative act $goba(x, y, \psi)$ is read as “I (x) am willing to decide to believe ψ.” The addressee can interpret this act as an affirmative answer to an oba; the offer to decide to believe a proposition is granted, hence the abbreviation goba. The communicative act $doba(x, y, \psi)$ is read as “I (x) am not willing to decide to believe ψ.” The addressee can interpret this act as a negative answer to an oba; the offer to decide to believe a proposition is denied, the abbreviation doba stands for denying an oba.

Example 6 (Dialogue about car insurance in Sesame Street). In the first line of the dialogue in Example 1, Tv states that its car is a Ferrari. We consider this expression equal to “Are you, Ia, willing to decide to believe that it is true that my car is a Ferrari?” allowing it to be translated to $oba(\mathsf{tv}, \mathsf{ia}, \mathsf{is\_a(this\_car, ferrari):t})$. In response, the Ia decides Tv’s offer in line 2. The Ia utters ‘Ok.’ which is interpreted to be equal to expression “I am willing to decide to believe that your car is a Ferrari.” In line 4, the Ia rejects Tv’s offer from line 3, the expression “I don’t believe that.” is interpreted as “no, I am not willing to decide to believe that it is true that your car is safe.” A rendition of the dialogue from Example 1 is presented next and used in the remainder of this paper.

1. $oba(\mathsf{tv}, \mathsf{ia}, \mathsf{is\_a(this\_car, ferrari):t})$
2. $goba(\mathsf{ia}, \mathsf{tv}, \mathsf{is\_a(this\_car, ferrari):t})$
3. $oba(\mathsf{tv}, \mathsf{ia}, \mathsf{is\_a(this\_car, safe):t})$
4. $doba(\mathsf{ia}, \mathsf{tv}, \mathsf{is\_a(this\_car, safe):t})$
5. $oba(\mathsf{ia}, \mathsf{tv}, \mathsf{is\_a(this\_car, safe):f})$
6. $doba(\mathsf{tv}, \mathsf{ia}, \mathsf{is\_a(this\_car, safe):f})$
7. $oba(\mathsf{ia}, \mathsf{tv}, \mathsf{is\_a(this\_car, ferrari):t} \rightarrowtail \mathsf{is\_a(this\_car, safe):f})$
8. $doba(\mathsf{tv}, \mathsf{ia}, \mathsf{is\_a(this\_car, ferrari):t} \rightarrowtail \mathsf{is\_a(this\_car, safe):f})$
9. $oba(\mathsf{ia}, \mathsf{tv}, \mathsf{a2d(ia, tv, is\_a(this\_car, safe):f, is\_a(this\_car, safe):t):t})$
10. $goba(\mathsf{tv}, \mathsf{ia}, \mathsf{a2d(tv, ia, is\_a(this\_car, safe):f, is\_a(this\_car, safe):t):t})$

3.2 Dialogue Game

A dialogue game is a finite set of dialogue rules that define when agents are allowed to communicate and how their cognitive states are to be updated afterwards. Similar to reasoning rules from Section 2.5, the pre and post-conditions of communicative acts are combined in dialogue rules to provide the semantics and the rules of usage of the communicative acts.

A generic dialogue rule of a communicative act $\lambda(x, y, \psi)$ states that if all pre-conditions of $\lambda(x, y, \psi)$ hold according to agent x's cognitive state, then $\lambda(x, y, \psi)$ may be uttered to agent y. The post-conditions define the contents of agent x and y's cognitive state after a communicative act is uttered or received.

$$CS_x \models pre(\lambda(x, y, \psi)) \quad \Longrightarrow \quad \{CS'_x, CS'_y\} \models post(\lambda(x, y, \psi)) \tag{DR}$$

3.3 Semantics of Communicative Acts

Information Offer. An agent's motivation to utter a question can be defined as balancing its belief and desire states [13]. Similarly, the motivation to offer information can be defined as an agent balancing her desire regarding another agent's belief sate and her belief state regarding this other agent's belief state. The dialogue rule for the communicative act of offering information is denoted OBA, which is an instantiation of the generic Dialogue Rule DR.

The motivation to offer information regarding proposition ψ is defined as the situation in which agent x has the desire that y believes ψ, and x is not aware that y already believes ψ. In addition, an agent is not allowed to put forward propositions she does not believe. This motivation is part of the pre-conditions to utter an offer.

$$(\psi \in D_x B_y),\ (\psi \notin B_x B_y),\ (\psi \in B_x) \ \in \ pre(oba(x, y, \psi))$$

An information offer is allowed by a speaker x to an addressee y if x is motivated to do so. Given these pre-conditions, addressee y may deduce the following properties of speaker x's cognitive state: x had the desire that y is to believe ψ, x was not aware that y believed ψ, and that the speaker x believes ψ. After the utterance of the offer, the cognitive state of the addressee y has changed according to the following post-conditions.

$$(\psi \in B_y D_x B_y),\ (\psi \in B_y I_x B_y),\ (\psi \in B_y B_x) \ \in \ post(oba(x, y, \psi))$$

A speaker may assume the addressee derives the same post-condition as she would have done if she had received the communicative act herself. Consequently, after uttering an $oba(x, y, \psi)$, speaker x is aware that y is aware that x desires that y believes ψ. In addition, the speaker x is aware that y is aware that x was not aware that y believed ψ, and that the speaker x is aware that the addressee y is aware that x believes ψ. The cognitive state of the speaker x has changed according to the following post-condition.

$$(\psi \in B_x B_y D_x B_y),\ (\psi \in B_x B_y I_x B_y),\ (\psi \in B_x B_y B_x) \ \in \ post(oba(x, y, \psi))$$

In addition to the motivations to communicate are the Gricean maxims that specify principles of cooperative dialogue [34]. These maxims state that utterances of communicative acts should be informative. For example, a speaker is not allowed to ask anything she already believes. Analogously, a speaker is not allowed to put forward information that the addressee already believes as seen from the speaker's perspective. In addition, agents are not allowed to utter communicative acts more than once because of the ideal communication channel in which no information is lost. To realize the restriction that an offer may not be uttered more than once, at least one of the previous post-conditions must not hold. This negated post-condition is added to the set of pre-conditions to restrict the situations in which offers may be uttered. (See Figure 2 for an overview of the pre and post-conditions.)

$$(\psi \notin B_x B_y D_x B_y) \;\in\; \mathit{pre}(\mathit{oba}(x, y, \psi))$$

Example 7 (Information offer). Abbreviate ψ for $\mathsf{is\_a(this\_car, ferrari){:}t}$. The communicative act $\mathit{oba}(\mathsf{tv}, \mathsf{ia}, \psi)$ is allowed in CS^1 (Example 3) with OBA rule resulting in CS^2.

$$\begin{aligned} CS^1_{\mathsf{tv}} &\models \{\, \psi \in D_{\mathsf{tv}} B_{\mathsf{ia}},\ \psi \notin B_{\mathsf{tv}} B_{\mathsf{ia}},\ \psi \in B_{\mathsf{tv}},\ \psi \notin B_{\mathsf{tv}} B_{\mathsf{ia}} D_{\mathsf{tv}} B_{\mathsf{ia}} \,\} \\ CS^2_{\mathsf{ia}} &\models \{\, \psi \in B_{\mathsf{ia}} D_{\mathsf{tv}} B_{\mathsf{ia}},\ \psi \in B_{\mathsf{ia}} I_{\mathsf{tv}} B_{\mathsf{ia}},\ \psi \in B_{\mathsf{ia}} B_{\mathsf{tv}} \,\} \\ CS^2_{\mathsf{tv}} &\models \{\, \psi \in B_{\mathsf{tv}} B_{\mathsf{ia}} D_{\mathsf{tv}} B_{\mathsf{ia}},\ \psi \in B_{\mathsf{tv}} B_{\mathsf{ia}} I_{\mathsf{tv}} B_{\mathsf{ia}},\ \psi \in B_{\mathsf{tv}} B_{\mathsf{ia}} B_{\mathsf{tv}} \,\} \end{aligned}$$

Granting of an Offer. Next to giving restrictions, the Gricean maxims provide motivations to answer questions and offers; in this case, offers should always be answered by either granting or declining it. The dialogue rule for the communicative act of goba is denoted GOBA, which is an instantiation of the generic Dialogue Rule DR.

An agent x is motivated to utter an accepting response $\mathit{goba}(x, y, \psi)$ if x is aware the addressee y has the desire to make x believe ψ, and, x believes ψ. This results in the following pre-conditions.

$$(\psi \in B_x D_y B_x),\ (\psi \in B_x) \;\in\; \mathit{pre}(\mathit{goba}(x, y, \psi))$$

Given these pre-conditions, addressee y may deduce the following properties of speaker x's cognitive state: x was aware that y desired that x believes ψ, and that x believes ψ. The set of post-conditions for the addressee's cognitive state are the following.

$$(\psi \in B_y B_x D_y B_x),\ (\psi \in B_y B_x) \;\in\; \mathit{post}(\mathit{goba}(x, y, \psi))$$

After the speaker has uttered a $\mathit{goba}(x, y, \psi)$, she may deduce the following properties of the addressee's cognitive state.

$$(\psi \in B_x B_y B_x D_y B_x),\ (\psi \in B_x B_y B_x) \;\in\; \mathit{post}(\mathit{goba}(x, y, \psi))$$

To prevent a superfluous goba from occurring, the speaker should not be aware that she uttered the communicative act before. Agents can be sure about this if at least on of the previous post-condition does not hold. The extra pre-condition reads as "agent x does not believe that y believes that x believes ψ".

$$(\psi \notin B_x B_y B_x) \;\in\; \mathit{pre}(\mathit{goba}(x, y, \psi))$$

Example 8 (Granting an oba). Abbreviate ψ for $\mathsf{is\_a(this\_car, ferrari){:}t}$. The communicative act $goba(\mathsf{tv}, \mathsf{ia}, \psi)$ is allowed in CS^2 (from Example 7) with GOBA rule resulting in CS^3. Example 5 shows that the Ia decides to believe ψ.

$$\begin{aligned} CS^2_{\mathsf{ia}} &\models \{\ \psi \in B_{\mathsf{ia}} D_{\mathsf{tv}} B_{\mathsf{ia}},\ \psi \in B_{\mathsf{ia}},\ \psi \notin B_{\mathsf{ia}} B_{\mathsf{tv}} B_{\mathsf{ia}}\ \} \\ CS^3_{\mathsf{tv}} &\models \{\ \psi \in B_{\mathsf{tv}} B_{\mathsf{ia}} D_{\mathsf{tv}} B_{\mathsf{ia}},\ \psi \in B_{\mathsf{tv}} B_{\mathsf{ia}}\ \} \\ CS^3_{\mathsf{ia}} &\models \{\ \psi \in B_{\mathsf{ia}} B_{\mathsf{tv}} B_{\mathsf{ia}} D_{\mathsf{tv}} B_{\mathsf{ia}},\ \psi \in B_{\mathsf{ia}} B_{\mathsf{tv}} B_{\mathsf{ia}}\ \} \end{aligned}$$

Declining an Offer. The motivation to utter a negative response $doba(x, y, \psi)$ to an information offer is similar to an affirmative response, with the main difference that the speaker in case of the negative response does not believe proposition ψ, while in the affirmative response she does. With Dialogue Rule DOBA we mean the generic Dialogue Rule DR instantiated for communicative act doba.

An agent x is motivated to utter a doba if the speaker x is aware the addressee y has the desire to make x believe ψ, and, x does not believe ψ. The preconditions are the following.

$$(\psi \in B_x D_y B_x),\ (\psi \notin B_x)\ \ \in\ pre(doba(x, y, \psi))$$

After the communicative act is uttered, the addressee of a doba may deduce properties of the speaker's cognitive state, and the speaker may deduce properties of the addressee's cognitive state. The set of post-conditions for speakers and addressees is the following.

$$\begin{matrix} (\psi \in B_y B_x D_y B_x),\ (\psi \in B_y I_x), \\ (\psi \in B_x B_y B_x D_y B_x),\ (\psi \in B_x B_y I_x) \end{matrix}\ \ \in\ post(doba(x, y, \psi))$$

To prevent a decline of an oba from being superfluous, at least one of the previous post-conditions must not hold. This criteria reads as "agent x does not believe that agent y believes that x is ignorant about ψ", which is the case if x has not informed y that she does not believe ψ.

$$(\psi \notin B_x B_y I_x)\ \ \in\ pre(doba(x, y, \psi))$$

Example 9 (Decline an oba). Abbreviate ψ for $\mathsf{is\_a(this\_car, safe){:}t}$. The communicative act $oba(\mathsf{tv}, \mathsf{ia}, \psi)$ is allowed in CS^3 (from Example 8) with OBA rule resulting in CS^4, from which $doba(\mathsf{ia}, \mathsf{tv}, \psi)$ is allowed with DOBA rule resulting in CS^5.

$$\begin{aligned} CS^4_{\mathsf{ia}} &\models \{\ \psi \in B_{\mathsf{ia}} D_{\mathsf{tv}} B_{\mathsf{ia}},\ \psi \notin B_{\mathsf{ia}},\ \psi \notin B_{\mathsf{ia}} B_{\mathsf{tv}} I_{\mathsf{ia}}\ \} \\ CS^5_{\mathsf{tv}} &\models \{\ \psi \in B_{\mathsf{tv}} B_{\mathsf{ia}} D_{\mathsf{tv}} B_{\mathsf{ia}},\ \psi \in B_{\mathsf{tv}} I_{\mathsf{ia}}\ \} \\ CS^5_{\mathsf{ia}} &\models \{\ \psi \in B_{\mathsf{ia}} B_{\mathsf{tv}} B_{\mathsf{ia}} D_{\mathsf{tv}} B_{\mathsf{ia}},\ \psi \in B_{\mathsf{ia}} B_{\mathsf{tv}} I_{\mathsf{ia}}\ \} \end{aligned}$$

Motivations to utter	Update of addressee y	Update of speaker x	Restriction to utter
$pre(oba(x,y,\psi))$ $\psi \in D_x B_y$ $\psi \notin B_x B_y$ $\psi \in B_x$	$post(oba(x,y,\psi))$ $\psi \in B_y D_x B_y$ $\psi \in B_y I_x B_y$ $\psi \in B_y B_x$	$post(oba(x,y,\psi))$ $\psi \in B_x B_y D_x B_y$ $\psi \in B_x B_y I_x B_y$ $\psi \in B_x B_y B_x$	$pre(oba(x,y,\psi))$ $\psi \notin B_x B_y D_x B_y$
$pre(goba(x,y,\psi))$ $\psi \in B_x D_y B_x$ $\psi \in B_x$	$post(goba(x,y,\psi))$ $\psi \in B_y B_x D_y B_x$ $\psi \in B_y B_x$	$post(goba(x,y,\psi))$ $\psi \in B_x B_y B_x D_y B_x$ $\psi \in B_x B_y B_x$	$pre(goba(x,y,\psi))$ $\psi \notin B_x B_y B_x$
$pre(doba(x,y,\psi))$ $\psi \in B_x D_y B_x$ $\psi \notin B_x$	$post(doba(x,y,\psi))$ $\psi \in B_y B_x D_y B_x$ $\psi \in B_y I_x$	$post(doba(x,y,\psi))$ $\psi \in B_x B_y B_x D_y B_x$ $\psi \in B_x B_y I_x$	$pre(doba(x,y,\psi))$ $\psi \notin B_x B_y I_x$

Fig. 2. Overview of the pre and post-conditions of the communicative acts

Auxiliary Offer. An auxiliary offer is an information offer that substantiates some claim to believe another proposition. This offer is syntactically indistinguishable from the offer defined in Section 3.3. However, from a semantic perspective the auxiliary offer is a different communicative act, since it has different pre-conditions. Nevertheless, the post-conditions derived from the pre-conditions are not different from those of the ordinary offer. To distinguish between the two offers, auxiliary offers are indexed 2. The dialogue rule is denoted OBA$_2$.

An agent x may utter an auxiliary offer if she has the desire that another agent y believes some proposition φ, and she is not aware that y already believes φ. These pre-conditions are equal to the motivation of the ordinary offer; however, a number of other pre-conditions are added. Agent x is motivated to utter an auxiliary offer regarding another proposition ψ if according to x, agent y would decide to believe φ if y decides to believe ψ. Agents use Reasoning Rule D2A$_1$ to deduce properties of other agent's cognitive state, and based on these findings justify auxiliary offers. Formally, if ψ is set-theoretically added to mental state $B_x B_y$, and φ is an element of the closure under the agent y's reasoning rules, then the auxiliary offer is allowed, that is, $\varphi \in Cn^r(B_x B_y \cup \{\psi\}, \mathrm{D2A}_1)$. In addition, agent x should believe ψ, and she should not be aware that y believes ψ.

$$\begin{array}{l}(\varphi \in D_x B_y),\ (\varphi \notin B_x B_y),\ (\psi \in B_x),\\ (\psi \notin B_x B_y),\ (\varphi \in Cn^r(B_x B_y \cup \{\psi\}, \mathrm{D2A}_1))\end{array} \in\ pre(oba_2(x,y,\psi))$$

Example 10 (Auxiliary offer). Abbreviate ψ for $\mathsf{is\_a(this\_car, safe){:}f}$ and φ for $\mathsf{is\_a(this\_car, ferrari){:}t} \rightarrowtail \mathsf{is\_a(this\_car, safe){:}f}$. The communicative act $doba(\mathsf{ia}, \mathsf{tv}, \psi)$ is allowed in CS^5_{ia} (from Example 9) with DOBA rule resulting in CS^6_{tv}, from which $oba(\mathsf{ia}, \mathsf{tv}, \varphi)$ is allowed with OBA$_2$ rule resulting in CS^7.

$$\begin{aligned} CS^6_{\mathsf{tv}} &\models \{\ \psi \in D_{\mathsf{tv}} B_{\mathsf{ia}},\ \psi \notin B_{\mathsf{tv}} B_{\mathsf{ia}},\ \varphi \in B_{\mathsf{tv}},\ \varphi \notin B_{\mathsf{tv}} B_{\mathsf{ia}},\\ &\qquad \psi \in Cn(B_{\mathsf{tv}} B_{\mathsf{ia}} \cup \{\varphi\}, \mathrm{D2A}_1)\ \}\\ CS^7_{\mathsf{tv}} &\models \{\ \psi \in B_{\mathsf{tv}} D_{\mathsf{ia}} B_{\mathsf{tv}},\ \psi \in B_{\mathsf{tv}} I_{\mathsf{ia}} B_{\mathsf{tv}},\ \psi \in B_{\mathsf{tv}} B_{\mathsf{ia}}\ \}\\ CS^7_{\mathsf{ia}} &\models \{\ \psi \in B_{\mathsf{ia}} B_{\mathsf{tv}} D_{\mathsf{ia}} B_{\mathsf{tv}},\ \psi \in B_{\mathsf{ia}} B_{\mathsf{tv}} I_{\mathsf{ia}} B_{\mathsf{tv}},\ \psi \in B_{\mathsf{ia}} B_{\mathsf{tv}} B_{\mathsf{ia}}\ \} \end{aligned}$$

4 To Agree to Disagree

Agents in conversation may become aware of parts of their communication partner's cognitive states, and while they do, it may happen that they become aware of irresolvable disagreements. If agents participate in some group-plan that requires mutual agreement on certain proposition, agents have a direct incentive to resolve disagreements regarding these propositions. Although we do not provide an explicit incentive, we do assume that agents have one, and, consequently, they will act to resolve disagreements.

A disagreement can be resolved with four different dialogue games: (1) an agent can convince the agent she disagrees with to believe a proposition that resolves the disagreement. (2) An agent can ask others to help to convince her to believe propositions that resolve the disagreement. (3) An agent can request the agents she disagrees with to forget propositions that result in the disagreement, and (4) an agent can ask others to help to convince her to forget propositions that result in the disagreement. We only consider the first situation: agents can resolve disagreements by convincing others to decide to believe propositions, thereby resolving the disagreement.

If all options to resolve the situation have been exhausted, agents are to conclude that they have an irresolvable disagreement about a specific proposition. If agents offer this awareness to the other agents, both can agree on their disagreement and make the disagreement a manifested belief. This agreement to disagree may trigger a new dialogue in which, for example, a coin flipping method is proposed to resolve the situation, or the meaning of the formula in the proposition is debated. A reasoning rule is defined in Section 4.3 to conclude that an agreement to disagree is in order. This rule combined with the dialogue game to offer information enables agents to agree to disagree.

4.1 Disagreements

Two pieces of information are conflicting when they are not subsumed under each other in the information order. That is, truth-values are in conflict, denoted $\ncong$, when they are unrelated in $\leq_k$. That is, $\theta_1 \ncong \theta_2$ iff $\theta_1 \nleq_k \theta_2$ and $\theta_2 \nleq_k \theta_1$.

A disagreement between agents x and y about formula p exists if and only if x believes $p{:}\theta_1$ and y believes $p{:}\theta_2$, and the truth-values are in conflict. Additionally, it needs to be the case that both propositions are the most informative, that is: for all truth-values θ_3 part of the bilattice hold that $\theta_3 \leq_k \theta_1$ and $\theta_3 \leq_k \theta_2$, for agents x and y respectively. Note that in a four-valued logic only one disagreement exists: *true* disagrees with *false* because $\mathsf{t} \nleq_k \mathsf{f}$, and $\mathsf{f} \nleq_k \mathsf{t}$. If an agent believes $p{:}\mathsf{u}$ and another believes $p{:}\mathsf{t}$ (and these propositions are the most informative), then they do not disagree about p, the latter agent is just more informed than the former naïve agent.

If a disagreement exists between two agents, both need not be aware of this. An agent x is aware she has a disagreement with another agent y if and only if she believes a proposition $p{:}\theta_1$ and she believes that y believes $p{:}\theta_2$ and the truth-values θ_1 and θ_2 are in conflict. A second-order disagreement awareness exists when an agent x is aware that another agent y believes a proposition $p{:}\theta_1$ and x is aware that y is aware that x believes proposition $p:\theta_2$ and θ_1 is in conflict with θ_2.

Example 11 (Disagreement awareness). Abbreviate p for $\mathsf{is\_a}(\mathsf{this\_car}, \mathsf{safe})$. In CS^3_{ia} (from Example 8, or Example 6, line 3), the Ia believes that Tv believes $p{:}\mathsf{t}$. From this moment, the Ia is aware of a disagreement about p: $CS^3_{\mathsf{ia}} \models \{p{:}\mathsf{f} \in B_{\mathsf{ia}},\ p{:}\mathsf{t} \in B_{\mathsf{ia}}B_{\mathsf{tv}}\}$. In CS^4_{tv} (from Example 9; or Example 6, line 4) that Tv also becomes aware of the disagreement: $CS^4_{\mathsf{tv}} \models \{p{:}\mathsf{t} \in B_{\mathsf{tv}},\ p{:}\mathsf{f} \in B_{\mathsf{tv}}B_{\mathsf{ia}}\}$. In Example 6, line 5, the Ia becomes aware of a second-order disagreement after she stated $p{:}\mathsf{f}$, because $CS^5_{\mathsf{ia}} \models \{p{:}\mathsf{f} \in B_{\mathsf{ia}}B_{\mathsf{tv}}B_{\mathsf{ia}},\ p{:}\mathsf{t} \in B_{\mathsf{ia}}B_{\mathsf{tv}}\}$. It is only after line 6 that Tv also becomes aware of this disagreement.

4.2 Resolving Disagreements

Assume there is a disagreement between agent x and y about a formula p with $p{:}\theta_1 \in B_x$ and $p{:}\theta_2 \in B_xB_y$. Proposition $p{:}\xi_1$ resolves the disagreement (viewed from x's perspective), if y would decide to believe $p{:}\xi_1$, and, as a result, y would decide to believe $p{:}\theta_1$ (viewed from x's perspective) due to Reasoning Rule D2A$_1$. We have:

$$p{:}\theta_1 \in Cn^r(B_xB_y \cup \{p{:}\xi_1\}, \mathrm{D2A}_1)$$

The previous proposition resolves the disagreement from x's perspective if y is to decide to believe the proposition. The following proposition resolves the situation from x's perspective if x herself decides to believe the proposition. Proposition $p:\xi_2$ resolves the disagreement, if x would decide to believe $p:\xi_2$, and, as a result, x would believe $p{:}\theta_2$ (due to Reasoning Rule D2A$_1$), thereby resolving the disagreement. Formally, we have:

$$p{:}\theta_2 \in Cn^r(B_xB_yB_x \cup \{p{:}\xi_2\}, \mathrm{D2A}_1)$$

An agent is only interested in the least informative proposition to resolve the situation, that is, the proposition with a truth-value that is the lower bound with respect to $\leq_k$. Remember that in the current dialogue game only additions of information are possible; consequently, resolving a disagreement can only take place by adding sufficient information to one of the two agents' cognitive states, rendering it inconsistent.

4.3 Reasoning Rule to Agree to Disagree

The pre-conditions are given next that state when agents are allowed to decide to believe that they agree to disagree. The reasoning rule to become aware of irresolvable disagreements is denoted D2A$_3$ which is an instantiation of the generic Reasoning Rule RR. The decision is denoted $d2a_3(x, a2d(x, y, p{:}\theta_1, p{:}\theta_2))$, we abbreviate proposition $a2d(x, y, p\theta_1, p{:}\theta_2)$ with κ for convenience. κ denotes that agent x and y agree to disagree on formula p. In the following paragraphs, we call agent x 'I' and y 'you'. The pre-conditions are the following.

1. I am aware that I have a disagreement with you about formula p.

$$(p{:}\theta_1 \in B_x),\ (p{:}\theta_2 \in B_xB_y),\ (\theta_1 \not\cong \theta_2)\ \in\ pre(d2a_3(x,\kappa))$$

2. I am aware that you are also aware of the disagreement.

$$(p{:}\theta_2 \in B_xB_y),\ (p{:}\theta_3 \in B_xB_yB_x),\ (\theta_3 \not\cong \theta_2)\ \in\ pre(d2a_3(x,\kappa))$$

3. I do *not* believe a set of propositions $\varPhi \subseteq B_x$ that I have not offered to you before and that could have resolved the disagreement if you had decided to believe them. Suppose $p{:}\xi_1$ is a proposition that if you had added it to your belief state, then the disagreement would have been resolved. For all sets of beliefs $\varPhi$ that if you had decided to believe them, then this would have resolved the disagreement, that is, $p{:}\xi_1 \in Cn^r(B_xB_y \cup \varPhi, \mathrm{D2A}_1)$. However, I have already offered $\varPhi$ to you, that is, the post-conditions of an oba apply, $\varPhi \subseteq B_xB_yD_xB_y$. In this situation I have no methods left (sets of propositions $\varPhi$) to persuade you.

$$\begin{matrix}\big(\forall\varPhi \subseteq B_x\ (p{:}\xi_1 \in Cn^r(B_xB_y \cup \varPhi, \mathrm{D2A}_1)\ \Rightarrow \\ \qquad\qquad \varPhi \subseteq B_xB_yD_xB_y)\big)\end{matrix}\ \in\ pre(d2a_3(x,\kappa))$$

4. I am aware that you do *not* believe a set of propositions $\varPsi \subseteq B_xB_y$ that you have not offered to me before that could have resolved the disagreement if I had decided to believe them. Suppose $p{:}\xi_2$ is the proposition that if I added it to my belief state, then the disagreement had been resolved. For all sets of beliefs $\varPsi$ that if they had been accepted by me, then this would have resolved the disagreement, $\psi{:}\xi_2 \in Cn^r(B_xB_yB_x \cup \varPsi, \mathrm{D2A}_1)$. However $\varPsi$ has already been offered to me, and I seem to have responded negative, that is, the post-conditions of a doba apply, $\varPsi \subseteq B_xB_yI_x$. In this situation I think that you have no methods (sets of propositions $\varPsi$) left to resolve the situation.

$$\begin{matrix}\big(\forall\varPsi \subseteq B_xB_y\ (p{:}\xi_2 \in Cn^r(B_xB_yB_x \cup \varPsi, \mathrm{D2A}_1)\ \Rightarrow \\ \qquad\qquad \varPsi \subseteq B_xB_yI_x)\big)\end{matrix}\ \in\ pre(d2a_3(x,\kappa))$$

These criteria are the pre-conditions for the reasoning rule to decide to add to a belief state that an agent is stuck in an irresolvable disagreement. If the agent has used the reasoning rule, the post-conditions hold that she actually believes that she agrees to disagree, and that she desires that the agent she disagrees with also believes this proposition.

$$(\kappa{:}\mathsf{t} \in B_x),\ (\kappa{:}\mathsf{t} \in D_xB_y)\ \in\ post(d2a_3(x,\kappa), B_x)$$

The dialogue game to offer information takes care of the communication of κ to y, and possibly reaching an actual agreement on this proposition, making the agreement to disagree common belief.

Example 12 (the Ia and Tv agree to disagree). Abbreviate p for $\mathsf{is\_a}(\mathsf{this\_car}, \mathsf{safe})$, ψ for $\mathsf{is\_a}(\mathsf{this\_car}, \mathsf{ferrari}){:}\mathsf{t}$ and κ for $a2d(\mathsf{ia}, \mathsf{tv}, p{:}\mathsf{t}, p{:}\mathsf{f})$. The communicative act

$doba(\mathsf{ia}, \mathsf{tv}, \psi \rightarrowtail p{:}\mathsf{f})$ is allowed in CS^8_{ia} (from Example 10) with DOBA rule resulting in CS^9_{tv} from which $oba(\mathsf{ia}, \mathsf{tv}, \kappa)$ is allowed with OBA rule resulting in CS^10. In the latter state, Tv is allowed to utter $goba(\mathsf{tv}, \mathsf{ia}, \kappa)$.

$$\begin{aligned}
CS^8_{\mathsf{tv}} &\models \{\ \psi \in D_{\mathsf{tv}}B_{\mathsf{ia}},\ \psi \notin B_{\mathsf{tv}}B_{\mathsf{ia}},\ \varphi \in B_{\mathsf{tv}},\ \varphi \notin B_{\mathsf{tv}}B_{\mathsf{ia}},\\
&\qquad \psi \in Cn^r(B_{\mathsf{tv}}B_{\mathsf{ia}} \cup \{\varphi\}, \mathrm{D2A}_1)\ \}\\
CS^9_{\mathsf{tv}} &\models \{\ \psi \in B_{\mathsf{tv}}D_{\mathsf{ia}}B_{\mathsf{tv}},\ \psi \in B_{\mathsf{tv}}I_{\mathsf{ia}}B_{\mathsf{tv}},\ \psi \in B_{\mathsf{tv}}B_{\mathsf{ia}}\ \}\\
CS^9_{\mathsf{ia}} &\models \{\ \psi \in B_{\mathsf{ia}}B_{\mathsf{tv}}D_{\mathsf{ia}}B_{\mathsf{tv}},\ \psi \in B_{\mathsf{ia}}B_{\mathsf{tv}}I_{\mathsf{ia}}B_{\mathsf{tv}},\ \psi \in B_{\mathsf{ia}}B_{\mathsf{tv}}B_{\mathsf{ia}}\ \}
\end{aligned}$$

5 Multi-agent System Architecture

The agent's mental states, reasoning and dialogue games are implemented in SWI-Prolog [35], resulting in flexible multi-agent system architectures. Other reasoning games specifying, for example, when agents are allowed to decide to forget propositions, or dialogue games specifying the semantics of posing and answering questions [14] can be added without the need to change the rules of existing games.

In Section 5.1, the order of execution of the different rules and choices made by agents are described in the agent's deliberation cycle. In Section 5.2, implementations are presented of mental states (MVL theories), reasoning and dialogue rules, which, in Section 5.3, result in a reasoning and dialogue space.

5.1 The Agent's Deliberation Cycle

The agent's deliberation cycle consists of three choices and the execution of five rules, see Figure 3 for a graphical depiction.

1. Check whether cognitive reasoning rules are applicable, select a rule and go to step 2 to execute this rule. If there are no applicable rules, go to step 4 to see whether the agent has received communicative acts.
2. Execute the cognitive reasoning rule that has been found in step 1. The executions of cognitive rules have no observable effects for other agents. Go to step 3 to update the agent's cognitive state accordingly.
3. Execute the appropriate update rule for reasoning rule from step 2. Go to step 1 to check whether more reasoning needs to be done.
4. Check whether communicative acts are received, that is, acts that are directed at the agent. Take the oldest act from the queue of received acts, and go to step 5. If the queue of received acts is empty, go to step 6 to check whether the agent is allowed to utter a communicative act.
5. Execute the appropriate update rule for the received communicative act from step 4. Go to step 1 to check whether reasoning can be done.
6. Check whether dialogue rules are applicable, select a rule and go to step 7 to execute this rule. If there are no applicable rules, go to step 4 to check whether communicative acts have been received.

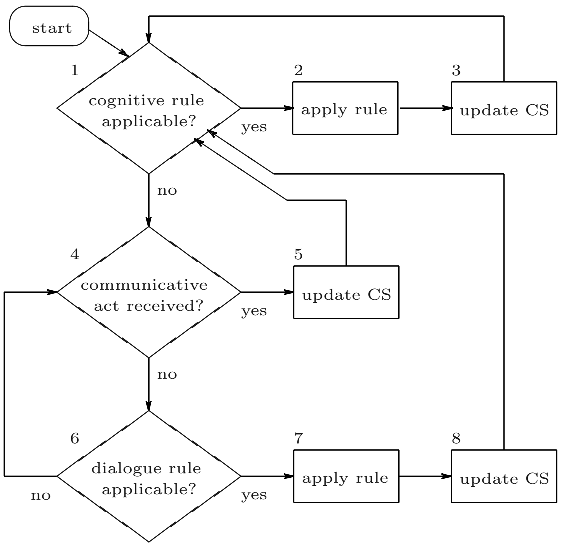

start
1
cognitive rule applicable?
yes
no
2
apply rule
3
update CS
4
communicative act received?
yes
no
5
update CS
6
dialogue rule applicable?
yes
no
7
apply rule
8
update CS

Fig. 3. The agent's deliberation cycle

7. Execute the dialogue rule that has been selected in step 6. The executions have the effect of uttering a communicative act directed at some other agent. Go step 8 to update the agent's cognitive state accordingly.
8. Execute the appropriate update rule for the uttered communicative act from step 7. Go to step 1 to check whether reasoning needs to be done.

Cycle 1-2-3 enforce that an agent's reasoning is done before she engages in conversation; her cognitive state is closed under reasoning before she is to test whether she received communicative acts, that is, $CS_x = Cn^r(CS_x, \mathcal{R})$. Cycle 1-4-5 enforces that all received communicative acts are processed and that the agent's cognitive state is updated accordingly before the agent is to test whether she is can utter communicative acts.

5.2 Implementation in Prolog

Remember that an agent's cognitive state consists of a number of mental states and that these states are theories of MVL. The programming language Prolog, and in particular SWI-Prolog [35], is used to implement the agent's cognitive states, the reasoning and dialogue rules.

A Prolog database is used to store the propositions part of the different theories that compose an agent's cognitive state; the Prolog inference engine is used to enforce that theories are closed under deduction rules as set forward in Definition 2. A Prolog term `prop(T, F, Tv)` states that formula `F` in theory `T` has at least truth-value `Tv`, that is, `F:Tv` $\in$ `T`. The test whether a proposition is part of a theory is implemented as a Prolog call to the database with the corresponding Prolog term for the proposition.

The following Prolog clauses implement complete theories of MVL. Clause `leq(k,`θ_1, θ_2`)` implements $\theta_1 \leq_k \theta_2$, and `oplus(k,` $\theta_1, \theta_2, \theta_3$`)` implements $\theta_3 = \theta_1 \oplus_k \theta_2$. The implementation of the bilattice structure is not presented.

```
prop(T, F, u).
prop(T, F, Tv1) :- prop(T, F, Tv2), leq(k, Tv1, Tv2).
prop(T, F, Tv1) :- prop(T, F, Tv2), prop(T, F, Tv3), \+ Tv2 = Tv3,
    oplus(k, Tv1, Tv2, Tv3).
```

Adding a proposition to the agent's cognitive state can be done straightforwardly by asserting the proposition. However, not all propositions need to be asserted, only those that cannot be derived from already asserted propositions.

```
add(T, F, Tv) :- \+ prop(T, F, Tv) -> assert(prop(T, F, Tv)); true.
```

Reasoning and dialogue rules are implemented by taking the conjunction of the pre-conditions of decisions and communicative acts as the body of a clause. The agent's reasoning capabilities are implemented with Prolog's deduction relation '`:-`'.

Example 13 (The Ia's Initial cognitive state in Prolog). Let `ms(b(ia))` denote B_{ia}, and `ms(d(ia), b(tv))` denote $D_{\mathsf{ia}}B_{\mathsf{tv}}$. From Example 3 we have:

```
prop(ms(d(ia),b(tv)), is_a(this_car, safe), f).
prop(ms(b(ia)), is_a(this_car, safe), f) :-
    prop(ms(b(ia)), is_a(this_car, ferrari), t).
```

The dialogue rule to offer information is implemented by taking the pre-conditions of the communicative act as the body of a Prolog clause. The update of the speaker and addressee's cognitive state are implemented as a sequence of actions of asserting propositions. Reasoning rules are implemented analogously.

```
dialogue_rule(oba(X, Y, F:Tv)) :-
    prop(ms(d(X), b(Y)), F, Tv),
 \+ prop(ms(b(X), b(Y)), F, Tv),
    prop(ms(b(X)), F, Tv),
 \+ prop(ms(b(X), b(Y), d(X), b(Y)), F, Tv).

update(oba(X, Y, F:Tv)) :-
    add(ms(b(Y), d(X), b(Y)), F, Tv),
    add(ms(b(Y), i(X), b(Y)), F, Tv),
    add(ms(b(X), b(Y), d(X), b(Y)), F, Tv),
    add(ms(b(X), b(Y), i(X), b(Y)), F, Tv).
```

5.3 Dialogue Space

The implementation of the reasoning and dialogue games provides a computational method to generate the space of all cognitive states reachable from an initial collective cognitive state. Although this space tends to become large for even a small number of agents, graphical depiction may give an intuitive feel whether protocols generate sensible communication. This space can be used by trace checkers to prove formal properties, like, for example, whether dialogues terminate in unique states (confluence property), or whether dialogues terminate at all (normalizing property).

The dialogue from Example 6 is a valid sequence of communicative acts in the dialogue game of offering information and reasoning game of deciding to believe propositions. From collective cognitive state from Example 3, the communicative acts of the dialogue are allowed, this is shown with Example 7, 8, 9, 10 and 12.

The complete space of valid dialogues in a dialogue game can be generated with the aid of software tools from an initial collective cognitive state. From the collective cognitive state in Example 3 the space of valid dialogues has been generated (not presented). The resulting graph has 37 nodes representing collective states and 66 edges representing utterances of communicative acts. This space comprises 177 different dialogues with three different final collective states. One has to remember that agents decide to believe propositions if these are consistent with their *current* belief state. This makes the timing of communicative acts of crucial importance, resulting in the three different endings, that is, the dialogue game does not have the confluent property. Resolving this non-confluence is part of future work.

6 Conclusion

In this paper, a formal semantics for an agent's cognitive state is given that allows agents to believe and desire inconsistent propositions. A reasoning rule is formulated enabling agents to decide to believe propositions if they are aware another agent also believes these propositions. Another reasoning rule is given in which agents decide to believe that they disagree with another agent and that this disagreement is irresolvable from both their perspectives.

A dialogue game is proposed for offering propositions and in particular the proposition that both agents agree to disagree. The semantics of the communicative acts are defined by formulating the rules of usage, being the pre-conditions that need to hold in the speaker's cognitive state, and the post-conditions that need to hold after the communicative act is uttered. With a dialogue game a formal system emerges in which sequences of communicative acts can be checked to be valid dialogues. In addition, dialogues spaces can be generated from dialogue rules, providing the possibility to analyse dialogue games on useful properties. One such property is whether unbalanced desire and belief states are resolved in the terminating cognitive states.

The agent's ability to become aware of irresolvable disagreements with some other agent combined with the ability to communicate this information, enables her to agree with the other that the disagreement is irresolvable. Both agents can then settle the disagreement with an agreement to disagree. Note that the agreement to disagree is based only on the cognitive state of the agents that actually have the disagreement.

Dialogue and reasoning games not only define the semantics of decisions and communicative acts, but also provide rules when to generate decisions and communicative acts. This allows straightforward Prolog implementations with intuitive design. Future research will address agents that strategically select which communicative acts to utter with the goal to arrive at a collective state in which desirable properties hold. Other research will centres around communicative acts for retracting information, that is: an offer to forget.

References

1. Parsons, S., Wooldridge, M., Amgoud, L.: Properties and complexity of some formal inter-agent dialogues. Journal of Logic and Computation **13** (2003) 347–376
2. Reed, C.A.: In Demazeau, Y., ed.: Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS 98), Paris, IEEE Press (1998) 246–253
3. Walton, D.N., Krabbe, E.C.W.: Commitment in Dialogue: Basic Concepts of Interpersonal Reasoning. SUNY Press. Albany, NY, USA (1995)
4. McBurney, P., Parsons, S.: Representing Epistemic Uncertainty by means of Dialectical Argumentation. Annals of Mathematics and Artificial Intelligence, Special Issue on Representations of Uncertainty **32** (2001) 125–169
5. Dignum, F., Dunin-Keplicz, B., Verbrugge, R.: Agent theory for team formation by dialogue. In Castelfranchi, C., Lesérance, Y., eds.: Pre-Proceedings of the Seventh International Workshop on Agent Theories, Architectures and Languages (ATAL 2000), Boston, USA (2000) 141–156
6. Prakken, H.: In Ojeda-Aciego, M., de Guzman, M.I.P., Brewka, G., Pereira, L.M., eds.: Proceedings of the 7th European Workshop on Logic for Artificial Intelligence (JELIA'00), Berlin, Germany, Springer Lecture Notes in AI 1919, Springer Verlag (2000) 224–238
7. McBurney, P., van Eijk, R.M., Parsons, S., Amgoud, L.: Journal of Autonomous Agents and Multi-Agent Systems **7** (2003) 232–273
8. Sadri, F., Toni, F., Rorroni, P.: Logic agents, dialogues and negotiation: and abductive approach. In Schroeder, M., Stathis, K., eds.: Proceedings of the Symposium on Information Agents for E-Commerce, Artificial Intelligence and the Simulation of Behaviour Conference (AISB 2001), York, UK, AISB (2001)
9. Hitchcock, D., McBurney, P., Parsons, S.: A framework for deliberation dialogues. In Hansen, H.V., Tindale, C.W., Blair, J.A., Johnson, R.H., eds.: Proceedings of the Fourth Biennial Conference on the Ontario Society of the Study of Argumentation (OSSA 2001), Windsor, Ontario, Canada (2001)
10. McBurney, P., Parsons, S.: Games that agents play: A formal framework for dialogues between autonomous agents. Journal of Logic, Language and Information **11** (2001) 315–334 Special Issue on Logic and Games.
11. Dignum, F., Dunin-Keplicz, B., Verbrugge, R.: Creating collective intentions through dialogue. Logic Journal of the IGPL **9** (2001) 305–319
12. Hulstijn, J.: Dialogue Models for Inquiry and Transaction. PhD thesis, Universiteit Twente, Enschede, The Netherlands (2000)
13. Beun, R.J.: On the Generation of Coherent Dialogue: A Computational Approach. Pragmatics & Cognition **9** (2001) 37–68
14. Lebbink, H.J., Witteman, C., Meyer, J.J.: Dialogue Games for Inconsistent and Biased Information. Electronic Lecture Notes of Theoretical Computer Science **52** (2003)
15. Labrou, Y.: Stand ardizing Agent Communication. LNCS **2086** (2001) 74–98

16. FIPA: Communicative act library specification. standard sc00037j. Technical report, Foundation for Intelligent Physical Agents (2002)
17. Searle, J.R.: Speech Acts: An Essay in the Philosophy of Language. Cambridge University Press, Cambridge, UK (1969)
18. Austin, J.L.: How To Do Things with Words. Harvard University Press, Cambridge Mass. (1962)
19. Audi, R.: Epistemology: a contemporary introduction to the theory of knowledge. Routledge (1998)
20. Latour, B.: Pandora's Hope: Essays on the Reality of Science Studies. Harvard University Press, Cambridge, Mass (1999)
21. Dummett, M.: What is a theory of meaning? (ii). In: The Seas of Language. Clarendon, Oxford, UK (1993) 34–93
22. Wright, C.: Strict finitism. In: Realism, Meaning and Truth. Blackwell, Oxford, UK (1993) 107–175
23. Coffa, J.A.: The Semantic Tradition from Kant to Carnap : To the Vienna Station. Cambridge University Press (1991)
24. Ellenbogen, S.: Wittgenstein's Account of Truth. SUNY series in philosophy. State University of New York Press (2003)
25. Wittgenstein, L.: Philosophical Investigation. Blackwell Publishers, UK (2001) first published in 1953.
26. Ginet, C.: Deciding to believe. In Steup, M., ed.: Knowledge, Truth, and Duty: Essays on Epistemic Justification, Responsibility, and Virtue. Oxford University Press (2001)
27. Rescher, N.: Many-valued Logic. McGraw-Hill (1969)
28. Butler, J.T.: Multiple-Valued Logic in VLSI. IEEE Computer Society Press (1991)
29. Meyer, J.J.C., van der Hoek, W.: Epistemic Logic for AI and Computer Science. Cambridge University Press (1995)
30. Ginsberg, M.L.: Multiv alued Logics: A Uniform Approach to Reasoning in Artificial Intelligence. Computational Intelligence **4** (1988) 265–316
31. Fitting, M.: Bilattices in logic programming. In Epstein, G., ed.: The Twentieth International Symposium on Multiple-Valued Logic, IEEE (1990) 238–246
32. Belnap Jr., N.D.: A useful four-valued logic. In Dunn, J.M., Epstein, G., eds.: Modern Uses of Multiple-Valued Logic, Dordrecht, D. Reidel (1977) 8–37
33. Sowa, J.F.: Knowledge Representation: Logical, Philosophical, and Computational Foundations. Brooks/Cole Pub Co (2000)
34. Grice, H.P.: Logic and conversation. In Cole, P., Morgan, J.L., eds.: Speech Acts. Volume 11 of Syntax and Semantics. Academic Press, New York, USA (1975) 41–58
35. Wielemaker, J.: Swi-prolog home page. `http://www.swi-prolog.org/` (2004)

Coordination of Complex Systems Based on Multi-agent Planning: Application to the Aircraft Simulation Domain

Frederic Marc[1,2], Amal El Fallah-Seghrouchni[2], and Irene Degirmenciyan-Cartault[1]

[1] Dassault Aviation, 78, quai Marcel dassault
92512 Saint Cloud Cedex France
[2] LIP6 - University of Paris 6,
8, Rue du Capitaine Scott,
75015 Paris France
{frederic.marc, irene.degirmenciyan}@dassault-aviation.fr
amal.elfallah@lip6.fr

Abstract. The design of complex systems needs sophisticated coordination mechanisms. This paper proposes an approach based on multi-agent planning to coordinate such systems in the tactical aircraft simulation domain. This paper presents an integral cycle, from modelling to validation, that enables the building of feasible multi-agent plans as developped in a project named SCALA promoted at Dassault Aviation.

1 Motivations

Multi-agent systems (MAS) are a suitable paradigm to design and model complex systems for well-known reasons like autonomy, reactivity, robustness, proactivity, etc. However an important issue remains the coordination and the cooperation of automomous agents in complex systems where these agents have to reorganize themselves dynamically. To cope with this issue, searchers have investigated several fields, especially multi-agent planning as a model for coordination and cooperation. Several approaches for multi-agent planning have been proposed such as [1] where a taxonomy of the relations between the agents plans and the development of a communication structure between autonomous agents are defined. Other approaches developed in [2, 3] are based on the paradigm of plan merging where each new agent entering in the system coordinates its plan with the current multi-agent plan. The notion of partial order between tasks has been exploited in [4]. The model allows the representation and the management of plans thanks to an extension of the Petri Net formalism to the Recursive Petri Nets [5].
Our application field concerns tactical aircraft simulation where the agents have to reorganize themselves when new events occur. This type of system can be seen as a complex system evolving in an open environment. In fact, we have to tackle the following problems: given an electronic institution where agents have

local goals and share global one's (*i.e.* aircraft domain), governed by social laws (described in a Graph of functionnal Dependencies), each agent of this institution has to plan autonomously its course of actions and coordinate its plan with the other agents. Then, in such framework, the allocation of tasks must be dynamic and context-dependent for a more flexible system. That is why when the agents of the institution reorganize themselves they have to take into account all the parameters that evolve during the simulation. These parameters may be the temporal objectives that the institution has to fulfil or the resources of the agents (for example: altitude, kerosene, etc.). The complexity of our system is basically inherent to the management of the resources and the dynamic aspect of the aircraft framework.

To deal with this complexity, we proposed in [6, 7] to model the different resources of the agents and the multi-agent plan starting from the well-used formalism of the hybrid automata [8] that we extended and enriched to take into account the flexibility of the agents' behaviour. The main advantage of this formalism is to model different clocks (a kind of variables) that evolve with different speeds. Thanks to the hybrid automata, we showed that it was possible to model a multi-agent plan as a network of hybrid automata where each hybrid automaton is a representation of an individual plan.

From this modelling, we expected to have a powerful formalism enabling to control the execution of the multi-agent plan and to validate it. Indeed, the control of the execution and the validation of multi-agent plans seem to be a real problematic now in this field of research. The most known work is realised in CIRCA [9] that proposes a system to control the safety of the system. For that, it develops "control plans" and sets of TAPs (Test/Action Pairs). Once scheduled, the verification of these TAPs ensure the system safety. The control plans are modelled by means of timed automata [10]. Our approach concerns the validation of the functionnal constraints of the mission more than the system safety. To sum up, CIRCA ensures that the system is safe under conditions and we would like to assure that the mission is feasible under conditions. To our knowledge, CIRCA is the only work that treats the control of multi-agent systems, in the aeronautical field, in the extension of the project named MASA-CIRCA [11].

However the flexibility offered by the designer makes the sets of constraints complex to be managed and thereby the control of the multi-agent plan. In fact, to enable flexible planning, the agents are not assumed to know *a priori* which course of actions to follow. We propose a dynamic management, via the sets of constraints, of the agents' plans. Thus allowing more flexibility inside the multi-agent plans. In addition, our approach provides the following mechanisms: to decompose the individual plans of the agents into a set of possible sub-plans; to extract the pertinent constraints to control the execution of the multi-agent plan and to validate it.

This paper is organized as follows: section 2 presents our domain of application namely Tactical Aircraft Simulation and the project SCALA (Co-operative System of Software Autonomous Agents). Section 3 presents our framework for the modelling of multi-agent plans as networks of synchronised hybrid automata from a Graph of Functionnal Dependencies (GFD). Section 4 introduces verification criteria in the multi-agent planning context. Section 5 describes the building cycle of feasible multi-agent plans. It presents the definitions and the computational mechanisms to control this cycle. Section 6 presents a distributed algorithm for multi-agent plan validation. Section 7 concludes this paper and introduces our future works.

2 SCALA and Tactical Aircraft Simulation

At Dassault-Aviation, we have been working on a project named SCALA which aims at proving the interest of a multi-agent approach to design and model complex systems, in particular in the framework of Tactical Aircraft Simulation. The major objectives carried out by SCALA is to make easier the modelling and the design of such systems. SCALA offers the designer a multi-agent based methodology providing a high level of abstraction. Different tools have been developped to rapidly setting up simulations by monitoring the behaviour of the system (individual and collective behaviour) and by using reusable mechanisms of cooperation. The behaviour is described by means of a Graph of Functionnal Dependencies (see section 3).

2.1 The Objectives of SCALA

This approach carries on several objectives to:

- Provide a tool to prototype multi-agent systems in proposing a methodology of design based on the functionnal requirements;
- Make easier the modelling and the design of such systems thanks to a high level description language;
- Simulate different types of organisation and communication protocols between agents;
- Constitute libraries of reusable mechanisms and protocols;
- Propose tools to support the design of the system and to monitor the behaviour oof the system.

2.2 The Methodology

The SCALA's methodology is based on a functionnal approach enabling the designers to elaborate and to model the whole behaviour of the multi-agent system. This methodology is composed of 8 steps :

1. Definition of the Graph of Functionnal Dependencies;
2. Definition of the Sub-Graphs (and the associated goals);
3. Identification of the expected relevant events;
4. Description of the elementary behaviours (or tasks) to be achieved by the agents;
5. Definition of the agents;

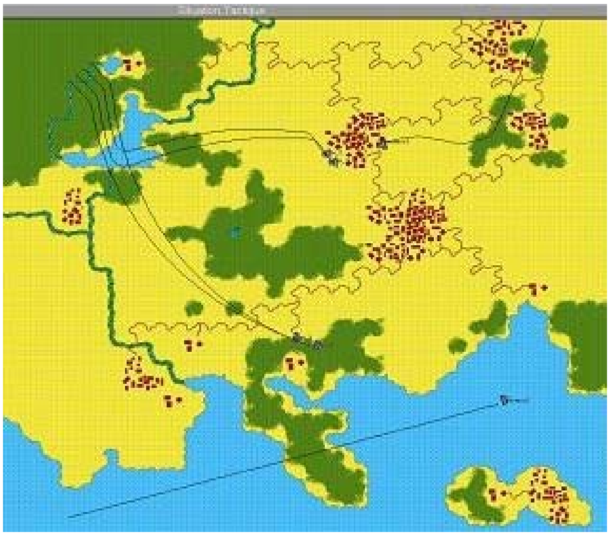


Fig. 1. The SCALA map

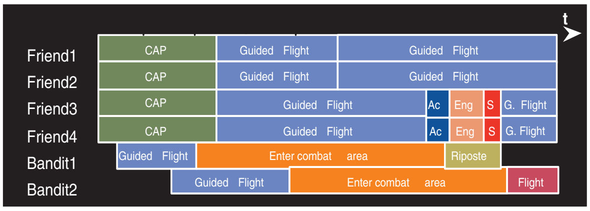


Fig. 2. The SCALA scheduler diagram

6. Definition of the groups of agents;
7. Choice of the social organisations;
8. Choice of the cooperation protocols.

To fulfill this methodology, we provide some tools in a first prototype of SCALA.

2.3 A First Implementation

A first prototype of SCALA has been developed using the JACK Agent Oriented Language software [12] based on Java. The following interception scenario of an aircraft mission has been developped with SCALA. These SCALA tools include:

- editors to design the agents and the groups of agents (skills, resources, flight plans of the planes, sensitivity to events...),
- a graph editor to easily design the GFDs (see section 3) which are associated to events, and also build a library of reusable behaviours,

– a map to situate the agents (figure 1),
– a scheduler diagram which represents the activities of the agents (figure 2).

3 Multi-agent Planning Based on Hybrid Automata

In our field of application, the designer expects a global coherent behaviour from the simulation. For that, this behaviour is specified through a GFD giving a decomposition of the global task into sub-tasks and their constraints.

3.1 The Graph of Functionnal Dependencies

The GFD helps the designer to model the global behaviour of the system at a given level of abstraction. More details about the GFD can be found in [7]. For a given task, the designer defines a graph of functionnal dependencies that is a decomposition of the global task into sub-tasks and provides the constraints between them. The graph of dependencies helps the designer to model the global behavior of the system at a given level of abstraction. In this modeling, a graph of functionnal dependencies (GFD) G is defined as a tuple $G =< Ta, S, To >$ where:

- Ta is a set of tasks t; $Ta = Ta_{elem} \biguplus Ta_{abs}$ where Ta_{elem} denotes the set of elementary tasks and Ta_{abs} denotes the set of abstract tasks (*i.e.* to be refined before their execution);
- S is the set of relations between the tasks within the graph defined as follows: $S = \{(t_i, t_j),\ t_i,\ t_j \in Ta/\ t_i\ r\ t_j,\ r \in \Re\ ||\ (r')^N t_i\ /\ r' \in \Re' = \Re - \{Ex\}\ ||\ (t_i)^N\}$ where $\Re = \{Ex, S_{start}, S_{end}, Seq\}$ and $N \in \mathbf{N}^*$ with the following semantics, we call these elements connectors in this paper:
  - $t_i\ Ex\ t_j$, the two tasks are exclusive, t_i inhibits t_j for ever;
  - $t_i\ S_{start}\ t_j$, the two tasks start at the same time;
  - $t_i\ S_{end}\ t_j$, the two tasks end at the same time;
  - $t_i\ Seq\ t_j$, t_i precedes t_j (used by default);
  - $(t_i)^N$ means that the task t_i must be executed N times;
  - $(r')^N\ t_i$ means that the task t_i is linked N times by a connector:
    - $(Seq)^N\ t_i$ corresponds to N cycles on task t_i: $t_i\ Seq\ t_i\ Seq\ ...\ Seq\ t_i$;
    - $(S_{start})^N\ t_i$ means that N tasks t_i start at the same time;
    - $(S_{end})^N\ t_i$ means that N tasks t_i end at the same time.
- To is a temporal objective associated with the graph G. To can be defined as a consistent conjunction of elementary temporal objectives. Let t_i be a task belonging to Ta and $goal_i$ a temporal objective on t_i: $To = \{(t_i, goal_i)\ /\ \mathcal{T}(t_i) = goal_i\}$. $\mathcal{T}()$ is the function which associates a temporal objective with a task.

3.2 An Example

This case study is introduced to show how a multi-agent graph can be modeled thanks to hybrid automata and developed in SCALA. Figure 3, we present

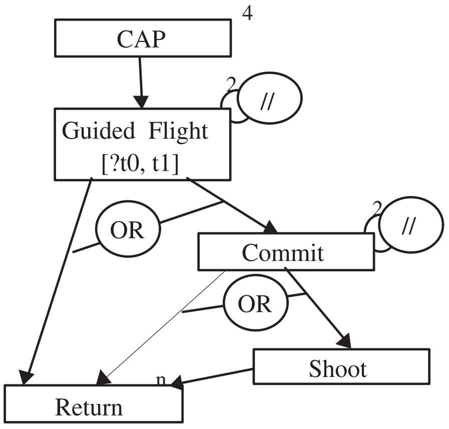


Fig. 3. Multi-agent Interception Graph

a graph of functionnal dependencies representing the expected behaviour of a mission of Threat Interception for a four aircraft division on alert on a CAP (Control Air Patrol). The mission of the aircraft is to protect the territory by intercepting the threats. The threat, in this case, represents an enemy plane entering in the friend area. The division begins the interception mission as soon as the bandit is detected.

The interception mission consists in several steps :

– The Guided Flight task follows the CAP as soon as a ”Contact” on the radar occurs.
– The cardinality of 2, associated to the Guided Flight task, means that two of the four aicraft have to start the interception phase, and moreover simultaneously (connector S_{start}). The temporal constraint [?t0, t1] on the Guided Flight task means that the section starts this task only if it can reach the threat before the date ”t1” (an approximation of the interception distance is computed).
– If the threat leaves the friend area, before the section of two aircraft has reached it at a distance ”d1”, the section returns to the base, else if the localization and the identification are OK, the section starts the commit (”Loc_ident” label in the automaton).
– As soon as the shoot is possible by one of the two friends, the Shoot task is performed (”Shoot_OK” label).
– Finally, the section returns to the base. The n associated to the Return task enables it to be performed by a varied number of aircraft at different moments of the mission.

3.3 Agent Plan Representation

In this part, we introduce the hybrid automata formalism. It is interesting since it allows the designer to take into account the agents resources (kerosene level, position, etc.) and the time as parameters of the simulation. In fact, the agents must be able to adapt their behaviours to cope with the changes of their resources and environment.

Motivation of Hybrid Automata. In order to model the different constraints, we need a formalism that could take into account different parameters and allow the representation of different properties of our system, such as :

– the reactivity : the system must be reactive in order to cope with the changes of the environment,
– the adaptibility : the system must manage different possible behaviours according to conditions of execution (*i.e.* the preconditions of a task),
– the dynamicity : the environment is dynamic and the variables of the system (agents and environment) may evolve during the simulation. These variables are, for example, the time and the different resources of the agents (kerosene, altitude, etc.),
– the synchronization : in our model, the entities have to synchronize themselves in order to achieve certain tasks (functionnal constraints).
– the validation : we want to use tools to validate the generated plans.

Moreover, a determinist behaviour is expected from the agents. It means that the agents have to execute certain task in a determinist way formalized by the designer.

A good candidate is the formalism of Hybrid Automata [8] since they allow to model different clocks that model the variables of the system even if they evolve with different speeds. This advantage is particularly suitable with our model. Furthermore, the hybrid automata keep all the properties of the automata. These properties concerns the model-checking, as follows :

– reachability : check if a certain state is reachable,
– security : under a set of conditions, guarantee that a certain situation never occurs,
– liveness : under a set of conditions, something will occur,
– deadlock absence : the automaton will never be in a situation in which it is impossible to go ahead.

A number of model-checkers already exists and we do not focus on this aspect in this paper; for example, HyTech [13].

Definition of the Hybrid Automata. The hybrid automata can be seen as an extension of timed automata [10]. A hybrid automaton is composed of two elements, a finite automaton and a set of clocks:

– A finite automaton A is a tuple:
$A =< Q, E, Tr, q_0, l >$ where Q is the set of finite states, E the set of labels,

  Tr the set of edges, q_0 the initial locations, and l an application associating the states of Q with elementary properties verified in this state.
- A set of clocks H, used to specify quantitative constraints associated with the edges. In hybrid automata, the clocks may evolve with different speeds.

The set Tr is a set of edge t such as $t \in Tr$,
$t =< s, (\{g\}, e, \{r\}), s' >$, where:

- s and s' are elements of Q, they model respectively the source and the target of the edge
  $t =< s, (\{g\}, e, \{r\}), s' >$ such that:
  - $\{g\}$ the set of guards, it represents the conditions on the clocks;
  - e the transition label, an element of E;
  - $\{r\}$ the set of actions on clocks.

The Reachable States. To represent a duration with an automaton, each task requires a *start state* and an *end state*. The set Q of reachable states is defined as follows. Let the graph $G =< Ta, S, To >$ and $t_i \in Ta$, for $1 \leq i \leq n$, a task of the graph G, t^i_{start} and t^i_{end}, respectively, the *start* and *end* states of the task:

$$\mathbf{Q} = \bigcup_{\mathbf{i=1}}^{\mathbf{n}} \{\mathbf{t^i_{\text{start}}}, \mathbf{t^i_{\text{end}}}\} \tag{1}$$

From this definition, we define two subsets Q_{start} and Q_{end} of Q, such as :

$$\mathbf{Q_{start}} = \{\mathbf{t^i_{\text{start}}}\}, \mathbf{Q_{end}} = \{\mathbf{t^i_{\text{end}}}\} \tag{2}$$

for $1 \leq i \leq n$. We obtain that $Q = Q_{start} \cup Q_{end}$. Let us note that the initial state is noted $q0$ and $q_0 \subset Q_{start}$.

The Labels of Transitions. The labels of transitions represent the conditions to move between tasks or to achieve them, that's why we introduce the notions of internal and external conditions of transition, respectively C_{in} and C_{ext}:

- The internal conditions enable to move from a start state to an end state of a task. This shift may correspond to the task execution or its interruption,
- The external conditions enable to move from a task to another, in other words from an end state to a start state of two tasks.

$$\mathbf{E} = \mathbf{C}_{\text{in}} \cup \mathbf{C}_{\text{ext}} \tag{3}$$

In our example, E is defined according to the predicates of the tasks composing the graph.

The Clocks. As said before, the hybrid automata enable the designer to model different clocks (*i.e.*variables of the system) evolving with different speeds. **The term of "clock" is used in the automata terminology but in our model, they represent the variables of the system**. In our case, two types of variables are considered:

– **The time**, we assume that agents are synchronized on a global clock since they act synchronously, represented by $\{clock\}_{A_i}$ (internal clock of the agent A_i) in the following equation. Hence, the only operations made on these local clocks are the tests specified through guards conditions.
– **The resources of the agents**, needed by the agents to achieve their plans, their evolution is managed through three operators: $use, consume$ and *produce*. They are modeled by $\{Res\}_{A_i}$: set of the resources owned by the agent A_i.

We had to use different clocks for simple reasons. The evolution laws of the variables are different so they cannot evolve with the same way. This cannot be modeled by Timed Automata for the previous reasons. Then, we have tried to model all the agents aspects and their resources because they are limiting factors in the simulation. In fact, the agents behaviour is a direct consequence of resources level. For example, an agent will shoot a target only if it owns at least one missile. This condition on the number of missile will appear on the guards of the task *Shoot*.

So, we can define H the set of clocks:

$$\mathbf{H} = \bigcup \{\mathbf{Res}\}_{\mathbf{A_i}} \cup \{\mathbf{clock}\}_{\mathbf{A_i}}, \tag{4}$$

for each $t_i \in Ta$. Let us remind our hypothesis of work, all the internal clocks of the agents are synchronized in a global clock. To our knowledge, there's no work considering different clocks that are not synchronized on a global clock (see [14]).

The Edges. The main advantage of the edges is to control the execution of the different plans and in our case the multi-agent plan. This type of control is very powerful because it enables to follow in real-time the evolutions of the variables of the system. There is two types of edge:

– Edges that connect 2 states ($t^i_{start} \rightarrow t^i_{end}$) of the same task, called in_{edge} (internal edges),
– Edges that connect 2 states ($t^i_{end} \rightarrow t^j_{start}$) of two different tasks, called ex_{edge} (external edges).

The in_{edge} edges model the *interruption* conditions of tasks, and the conditions on clocks to be fulfilled. They are labeled as follows:
Considering $tr \in in_{edge}$, $s \in Q_{start}$ and $s' \in Q_{end}$, s and s' respectively the source and the target of tr:

- The guard is the set of interruption conditions of the task X: $X_{interrupt}$,
- The label of transition $l_X \in C_{in}$ is the condition of achievement of the task X,
- The set of actions on the clocks from the post-conditions of the task X: X_{post}.

$$\mathbf{tr} = < \mathbf{s}, (\mathbf{X_{interrupt}}, \mathbf{l_X}, \mathbf{X_{post}}), \mathbf{s'} > \tag{5}$$

The ex_{edge} edges model the preconditions to be checked in order to execute a task. They are labeled as follows:
Considering $tr \in ex_{edge}$, $s \in Q_{end}$ and $s' \in Q_{start}$, s and s' respectively the source and the target of tr:

- The guard is the set of preconditions of X on the resources: X_{pre},
- The label of transition $l_X \in C_{ext}$ is the moving condition of X from s to s',
- No actions on clocks.

$$\mathbf{tr} = < \mathbf{s}, (\mathbf{X_{pre}}, \mathbf{l_X}, -), \mathbf{s'} > \tag{6}$$

3.4 Multi-agent Planning

In this section, we present the modelling of the multi-agent plans as networks of Synchronized Hybrid Automata [8].

The Synchronized Product. As we said before, we consider that each agent owns a plan that is modeled by an hybrid automaton. Consequently, the multi-agent plan is the synchronized product of each hybrid automaton. Let us define the synchronized product :
Considering n hybrid automata $A_i = < Q_i, E_i, Tr_i, q_{0,i}, l_i,$
$H_i >$, for $i = 1, ..., n$.
The *Cartesian product* $A_1 \times ... \times A_n$ of these automata is $A = < Q, E, Tr, q_0, l, H >$, such as:

- $Q = Q_1 \times ... \times Q_n$;
- $E = \prod_1^n (E_i \cup \{-\})$;
- $T = \{((q_1, ..., q_n), (e_1, ..., e_n), (q_1^{'}, ..., q_n^{'})|,$
  $e_i =' -'$ and $q_i^{'} = q_i$
  or $e_i \neq' -'$ and $(q_i, e_i, q_i^{'}) \in Tr_i\}$;
- $q_0 = (q_{0,1}, q_{0,2}, ..., q_{0,n})$;
- $l((q_1, ..., q_n)) = \bigcup_1^n l_i(q_i)$;
- $H = H_1 \times ... \times H_n$.

So, in this product, each automaton may do a local transition, or do nothing (empty action modeled by '-') during a transition. It is not necessary to synchronize all the transitions of all the automata. The synchronization consists of a set of *Synchronization* that label the transitions to be synchronized. Consequently, an execution of the synchronized product is an execution of the Cartesian product restricted to the label of transitions. In our case, we only synchronize the edges concerning the temporal connectors S_{start} and S_{end}.

Indeed the synchronization of individual agent's plans is done with respect to functionnal constraints and classical synchronisation technics of the automata formalism like "send / reception" messages. In our modelling, we map the different parameters of the synchronized automaton to the different notions of multi-agent planning. Let us remind that the synchronized automaton is a representation of all the planned actions of the agents. So, if we consider a synchronized automaton A, such as $A =< Q, E, Tr, q_0, l, H >$:

– Q represents the local goals of all the agents, that is to say all the reachable internal states of the agents;
– E models the high level label of transitions that are used in the GDF;
– Tr models the pre-, post- and interruption-conditions of the tasks. In fact, the different conditions on clocks enables to control the multi-agent plan execution and by the way, the agents behaviour. Furthermore, the *Synchronization* is realised by means of this set;
– q_0 is the set of initial states of all the agents acting for the same global goal;
– $l(q_1, ..., q_n)$ is the association of the elementary functions that maps the agents states to elementary properties verified in these states;
– H is the set of clocks of all the agents.

Two Levels of Synchronization. To guarantee the efficiency of our model, two levels of synchronization are distinguished:

– A *functionnal* level: modeled by the connectors S_{start}, S_{end} and Ex in the graph of dependencies. This level concerns the global behaviour of the system (*i.e.* the functionnal dependencies).
– An *internal* level: modeled by the connector *Seq*. This connector controls the evolution inside the graph of functionnal dependencies and implies important issues in the multi-agent context.

The *internal* level models the evolution process of the executed tasks in the graph. The evolution rule inside the graph may be the following:
"If two tasks X and Y are linked by the relation X *Seq* Y, Y should be executed iff X has already been executed".
This assumption may be generalized to a set of tasks where each task is linked by the *Seq* connector to another task, such as:
"If a set of tasks T_X and a task Y are linked such as $\forall x \in T_X$ / x *Seq* Y, Y should be executed iff all the tasks of T_X have already been executed".
To control this evolution, we use the "shared variables" technics to synchronize the internal level. For that, we add a variable (that will be shared by the agents) to each task that controls the state of task (executed or not). It means that a task will become feasible as soon as all the tasks in relation with a *Seq* connector will have been executed. This internal synchronization has an impact on the definition of the edges.
Considering a task X, and a set of tasks T_X (as defined below), the modifications are the following:

- for the ex_{edge} whose X is the target, the guards must check that all the tasks of T_X have been executed by checking all the variables associated to the tasks of the set T_X.
- for the in_{edge}, the value of the variable associated with X must become true to model that X has been executed.

This technics enables us synchronizing the individual automata at a first level of synchronization. We only propose, here, a first solution that should be improved and better formalized.

An Example of Synchronization Connector: The S_{start} Connector. Considering the following constraint between three tasks (X, Y and Z) in the graph of dependencies, S is defined as follows:

- $S_{XYZ} = \{Z\ Seq\ X,\ X\ S_{start}\ Y\}$

Let us assume that this part of the graph is executed by two agents A1 and A2 that have respectively two sets of tasks Ta_{A1} composed by Z and X, and Ta_{A2} composed by Y. The synchronization is carried out through three states Es,

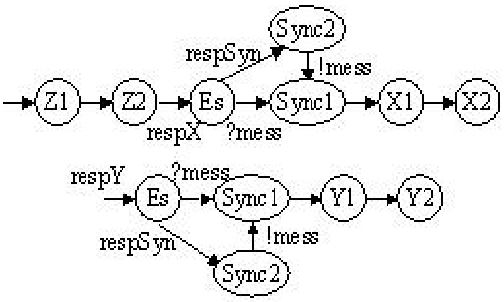


Fig. 4. S_{start} Connector for two automata

*Sync*1 and *Sync*2 added to the agents automata enabling a "synchronization by message sending". In that case, we obtain :

- $S_{XYZ,A1} = \{Z\ Seq\ X,\ X\ S_{start}\ Y^*\}$,
- $S_{XYZ,A2} = \{X^*\ S_{start}\ Y\}$.

An agent has to take into account some of other agent tasks (labeled with (*)), as constraints when building its automaton [15]. The agent, responsible for the synchronization, has its *respSyn* variable equals to true. This agent moves to the *Sync*1 state and sends a message (!*mess*) to the selected agents that are waiting for this message (?*mess*). As soon as the message has been sent and received, the agents start to execute their tasks. This kind of synchronization constitutes a sub-set of the *synchronization* set. For example, for two automata,

the transition $\{(<...,?mess,...>,<...,!mess,...>)\}$ belongs to the set Tr of the synchronized product. The use of this connector adds two new states to the set of reachable states of the agent.
Let us start with the automaton of agent A_i. The formula 1 provides Q^{A_i} as a set of reachable states. After the synchronization of A_i with A_j, the new set of reachable states $Q^{A_i}_{Synchro_{start}}$ is:

$$Q^{A_i}_{Synchro_{start}} = Q^{A_i} \cup \{Es_{A_i}, Sync1_{A_i}, Sync2_{A_i}\} \tag{7}$$

Incidence of the Operators on the Automata. We can sum up the modifications on the sets of reachable states Q and edges Tr of the agents A_i according to the following algorithm:

The Connector / Synchronization algorithm

```
program Synchro(Automaton at, Connector Connect)
begin
{Considering an automaton at composed by a set
Q(at) of reachable states and a set of
transition Tr(at) and a synchronization
Connector between two tasks Z and X}
Switch(Connect)
    case 'Sstart':
     Q(at) = Q(at) + {Es, Sync1, Sync2};
     Tr(at) = (Tr(at) - <Z2,(preX),X1>) +
     {<Z2, (-, respX, -), Es>,
     <Es, (-, respSyn, -),Sync2>,
     <Es, (-, ?mess, -), Sync1>,
     <Sync2, (-, !mess, -), Sync1>,
     <Sync1, (preX), X1>};

    case 'Send':
     Q(at) = Q(at) + {Sync1end, Sync2end};
     Tr(at) = (Tr(at) - <X1,(postX),X2>) +
     {<X1, (postX), Sync2end>,
     <X1, (-, ?mess, -), Sync1end>,
     <Sync2end, (-, !mess, -), X2>,
     <Sync1end, (postX), X2>};

    case 'Ex':
     If(Z, X belong to Ai)
      Q(at) = (Q(at) - {Z2}) + {Z21, Z22};
      {The set Tr(at) is built according
       to the conditions in the graph};
     End If
end.
```

An Example of Representation in Automata. In our case study, all the agents involved in the mission are able to perform all the tasks of the graph.

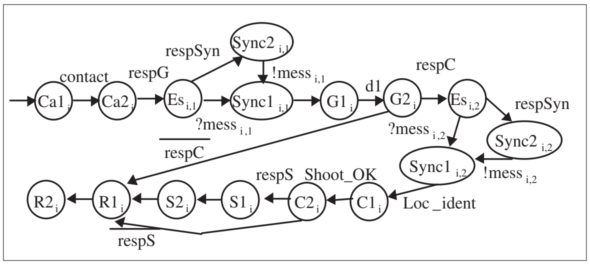


Fig. 5. Interception Automaton

So, the actions of each agent are described in a generic automaton that models, in this case, an individual version of the GFD. The branches of the automata are covered according to the context (the current state of the agents resources, the temporal and spatial constraints). An agent responsible of a task means that it is designated to perform it. This dynamic allocation is the result of a cooperation mechanism (ex: the choice of the two agents which leave the CAP) or due to the context (ex: the best placed agent shoots the threat: see *respS* in the automaton).

- $Ta = \{CAP, GuidedFlight_p, Commit_p, Shoot_p,$ $Return\}$,
- $S = \{(CAP)^4\ Seq\ (S_{start})^4 GuidedFlight_p,$ $(S_{start})^4 GuidedFlight_p\ Seq\ (S_{start})^2 Commit_p,$ $(Return)^n\ Ex\ (S_{start})^2 Commit_p,$ $(S_{start})^2 Commit_p\ Seq\ Shoot_p, Shoot_p\ Seq\ (Return)^n, (Return)^n\ Ex\ Shoot_p,$ $(CAP^4), (S_{start})^2 Commit_p, (S_{start})^4 GuidedFlight_p, (Return)^n\}$,
- $To = \{\mathcal{T}(GuidedFlight_p){=}t1\}$.

In our case, we obtain for an agent A_i:
$Q^{A_i} = \{CAP^i_{start}, CAP^i_{end}, GuidedFlight^i_{d,start}, GuidedFlight^i_{d,end}, Commit^i_{d,start}, Commit^i_{d,end}, Shoot^i_{d,start}, Shoot^i_{d,end}, Return^i_{start}, Return^i_{end}\}$ $\cup\{Es_{i,j}, Sync1_{i,j}, Sync2_{i,j}\}$ for $j \in [1, 2]$, j corresponds to the number, in this case, of start synchronization.
To make easier the writing of the automaton, we replace the previous states in the same order $Q^{A_i} = \{Ca1^i, Ca2^i, G1^i_p, G2^i_p, C1^i_p, C2^i_p, S1^i_p, S2^i_p, R1^i, R2^i\}\cup$ $\{Es_{i,j}, Sync1_{i,j}, Sync2_{i,j}\}$.

3.5 Propagation of the Constraints

As mentioned before, the formalism of hybrid automata enables to control the plan execution. To finalize their automata, the agents have to propagate the constraints about time and resources along the different branches of their automaton. There are two levels of propagation. Locally, the agents can propagate (by induction) the constraints on the **private** resources (physical resources). Globally, they have to propagate the constraints about temporal objectives (**shared** constraints) by synchronising themselves.

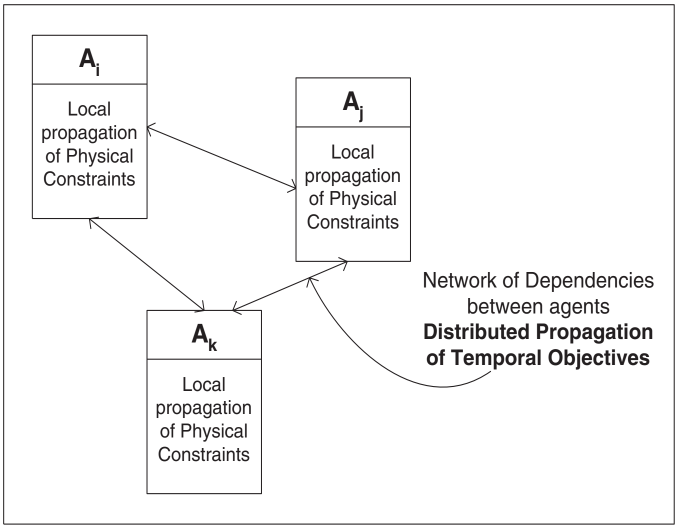

A_i
Local propagation of Physical Constraints
A_j
Local propagation of Physical Constraints
A_k
Local propagation of Physical Constraints
Network of Dependencies between agents
Distributed Propagation of Temporal Objectives

Fig. 6. Two Levels of Constraints Propagation

Moreover, we expect from the dynamic control to have a *preemptive* approach. We assume that is not necessary to carry on a plan if a temporal objective or a condition on the resources is violated. The control enables to anticipate on future problems that will inevitably occur. In that way, we can consider that we have an adaptative system because it will automatically reorganize itself to cope with these problems. The complexity is due to the part of flexibility introduced when designing the GFD (for instance the number of agents to execute a task is specified but not their allocation which is postponed). This flexibility implies for the agents to manage dynamically their courses of actions, to adapt the control to their environment (management of the constraints propagation) and to validate their choices.

4 Verification Criteria in Dynamic Context

In a multi-agent context, the control and the agents' choices should have some impacts on the multi-agent plan (MAP). The control of the MAP execution has an interest if it leads to a validation of the agents' choices and *a fortiori* the MAP. To validate the feasibility of the MAP, we have defined several criteria.

4.1 The Coverage

The coverage is an important point in the MAP validation. The agents have to ensure that all the tasks are allocated to the good number of agents. This approach is clearly related to the task-oriented approach. If the allocation is wrong, it will create a deadlock in the MAP execution. A protocol of validation deals with this problem when a reorganisation is needed.

Definition: The MAP is **C-compatible** if all the tasks of the graph are allocated to the sufficient number of agents.

Let P be a MAP; *Remaining*(P) the set of remaining tasks (that have not been executed) and So, for the current MAP P, the set of remaining tasks *Remaining*(P) (that have not been executed) and *Resp*(ti) the current number of agents responsible (if a task belongs to the agent's set of tasks) for the task ti, we say that:

$$\boxed{\begin{array}{c} P \text{ is C-compatible} \\ \text{Iff } \forall ti \in Remaining(P),\ Resp(ti) \geq card(ti) \end{array}}$$

To ensure a coherent allocation of the tasks, we developed an algorithm of functionnal allocation that takes into account the functionnal constraints between the tasks in the GFD and the skills of the agents. By lack of place, in this paper, we do not focus on this algorithm.

4.2 The Agents' Resources

The resources of the agents are locally controled. At any time of the simulation, the agents have to ensure that they have enough resources to execute their partition of the MAP (R-compatible). We propose in this paper a **mechanism to control** the MAP execution. We will come back to this mechanism on the next section.

4.3 The MAP Feasibility

The last point is the feasibilty of the MAP. Actually, we have to ensure that under the current allocation of tasks and the current set of plans of the agents, the MAP execution is possible. We have to check that in our context the current partitions of the agents enable to reach at least one objective of the MAP. Let us remind that the MAP may own several objectives but the agents do not have to reach all of them.

So, if we consider a global goal *Goal* of a MAP and Ta the set of tasks of the GFD, this goal is a conjunction of some elementary tasks such as:

$$\boxed{Goal = (ti \wedge tj \vee ... \vee tl) \text{ with } \forall ti \in Goal,\ ti \in Ta}$$

Definition: The current multi-agent plan is **Goal-compatible** if the agents can at least reach one of the objectives.

Considering P, the current MAP, the reachability of an objective X in P is noted: $P \Rightarrow X$. Let us note that an objective can be seen as an element of *Goal*. The condition of feasibility can be noted :

$$\boxed{\text{If } \exists X \in Goal \ / \ P \Rightarrow X \text{ Then } P \text{ is Goal-compatible}}$$

We see on the next section an algorithm to check the feasibility of the MAPs.

If the MAP does not recover these criteria, it is said to be in an unstable state. Actually, a MAP is validated if it is compatible at these three levels. In the case where the system cannot get to a safe state, it can reach a **"homestate"** (i.e. state in which the agents are able to reach again a safe state).

This type of states enables the system to go back on a stable situation. For example, in the case of a tactical aircraft simulation, if the patrol is no more able to treat a threat because the planes composing the patrol have no more missiles, a possible "homestate" can be reached trough the task "go back to the base".

5 Building Feasible Multi-agent Plans

We expect from this dynamic control to preempt future problems on the MAP execution. For that, the agents should consider all the possibilities of evolution according to their sets of tasks, and manage them at a local and global levels. To avoid the combinatory explosion, we introduce a reduction mechanism 5.3.

5.1 Definitions

Definition: The **Set of Possible Paths (SPP)** corresponds to all the possible paths the agents can plan according to their partially ordered set of tasks.

Definition: A **Feasible Path (FP)** is a path in the individual hybrid automata going from the initial state to the end state which satisfies all the initial constraints on the nodes of its path.

A FP is a tuple as $FP =< A, \{cont\}_Q >$ with A a classical hybrid automaton as defined in our formalism [7]; and $\{cont\}_Q$ the set of constraints on A on the state Q.

Definition: A **Set of Feasible Paths (SFP)** corresponds to all the possible paths planned individually by an agent satisfiying all the constraints locally propagated (private resources of the agents), $SFP \subset SPP$.

Definition: A **Control Set (CS)** is a set of pertinent constraints that enables the agents to control the flexibility of the multi-agent plan execution. Each agent computes its own CSs (as detailed in the next section).

Definition: Stable SFP For an agent A, if its control set **CS** is verified then its SFP, noted SFP^A is said to be stable, noted $SFP^A \rightarrow stable$.

Figure 7, the agent owns two possible FPs.

5.2 Computation of the Control Sets

To compute the control sets, we need the following features:

- the agent is in a state X
- the sub-set of possible FPs from the state X is noted: $SFP_{X\rightarrow}$
- the agent considers the sets of guards of all the FPs where it is able to evolve from state X, noted: $G_{X\rightarrow} = \cup\{cont\}_{X_i}$ with $i \in [1, card(SFP_{X\rightarrow})]$

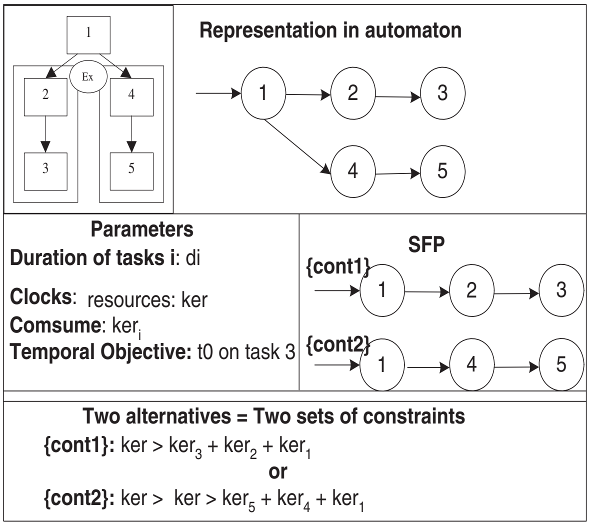

1
Ex
2
4
3
5
Representation in automaton
1
2
3
4
5
Parameters
Duration of tasks i: di
Clocks: resources: ker
Comsume: ker_i
Temporal Objective: t0 on task 3
SFP
{cont1}
1
2
3
{cont2}
1
4
5
Two alternatives = Two sets of constraints
{cont1}: ker > ker_3 + ker_2 + ker_1
or
{cont2}: ker > ker > ker_5 + ker_4 + ker_1

Fig. 7. Example of decomposition

Under these assumptions, a CS in a state X is noted CS_X and it is a set of most pertinent constraints for each clock considering all the remaining FPs in $SFP_{X\rightarrow}$:

$$CS_X = \bigcup_{clock \in G_{X\rightarrow}} prep(\bigcup_{FP_i} C_{i,clock}(G_{X\rightarrow})) \tag{8}$$

In fact, the agent considers all the future possible guards and merge the constraints on clocks by considering the most critical ones. We define the notion of **preponderancy** between constraints on a clock c such that:

- "$<$" : Considering $c_<=\{\forall\ (t < c) \in C_i(G_{X\rightarrow})\}$,
  prep$(c_<) = \{t < c/\forall t' \in c_<, t \leq t'\}$
- "$>$" : Considering $c_>=\{\forall\ (t > c) \in C_i(G_{X\rightarrow})\}$,
  prep$(c_>) = \{t > c/\forall t' \in c_>, t \geq t'\}$

In the example figure 7, a task i has a duration d_i and a kerosene consumption ker_i. The clock ker represents the level of kerosene. The automaton has two possible branches: $1 \rightarrow 2 \rightarrow 3$ or $1 \rightarrow 4 \rightarrow 5$. The execution of the task 3 is constrained by a temporal objective t_0.

If the agent in the state 1 wants to control its two possibilites of evolution and not focus on only one branch (this solution is too restrictive), it has to consider the two corresponding sets of constraints obtained on the first edge. So, to manage its flexibility, an agent has to concurrently manage its possible alternatives of evolution without favoring *a priori* one of them. This solution seems to be very costly if the agent checks the sets of constraints independantly. To simplify this process, we extract a single set of pertinent constraints. Let us note that we take into account the flexible behaviour of the agents and their dynamic choices of evolution (reactive planning).

In this example, if we consider that $ker_5 + ker_4 + ker_1$ is greater than $ker_3 + ker_2 + ker_1$, the most pertinent constraint is $ker > ker_5 + ker_4 + ker_1$ since it ensures that the agent in the state 1 can follow anyone of the two branches (*i. e.* the two branches are feasible paths).

5.3 Reduction of SFP

During the simulation, the agents choose their courses of actions according to the following three levels:

– functionnal: the agents evolve in the graph according to the functionnal constraints related in the graph of functionnal dependencies (*e.g.* constraints of synchronisation, exclusion, etc.),
– contextual: the agents may choose a course of actions according to the context (e.g. knowledge of the agents about their environment),
– control: the agents detect a violation of constraints. This level is complex to manage because the remaining tasks of the MAP have to be, in certain circumstances, re-allocated to the others.

The violation of the control sets makes the SFP unstable. The reduction of the SFP enables an agent to reach a stable SFP. To sum up, considering a set of agents, noted *Gp* :

Definition: A multi-agent plan is **R-compatible** (resources compatible) if at a date "t" all the agents of the *Gp* own a stable SFP.

So for a MAP *P*, this condition is validated under the following assumption:

$$\boxed{\begin{array}{c}\text{If } \forall A_i \in Gp,\ SFP^{A_i} \rightarrow stable \\ \text{Then } P \text{ is R-compatible}\end{array}}$$

According to their choices with respect to the functionnal constraints of the GFD, the result of the control and the context, the agents reduce their SFPs to only consider the remaining possible automata. Hence, they eliminate from their SFP the no more available automata as soon as they become impossible to execute. Even if the control is made at a local level by the agents, all the choices may have some impacts at the multi-agent behavioural level. We treat an example of such evolution in [16].

5.4 The Cycle for Building the Feasible MAP

To build the feasible MAP, our approach is composed of three main steps as shown in figure 8:

Step 1: The GFD is firstly allocated to the agents. So, they own a set of partially ordered tasks.

Step 2: The agents compose their **SFP** by planning all the possible paths considering their set of partially ordered tasks. At this step, the agents validate locally the feasibility of these paths by propagating the constraints on their resources.

Step 3: The agents synchronise their FP to obtain a set of feasible MAPs taking into account the temporal objectives. These MAPs correspond to networks of synchronised automata.

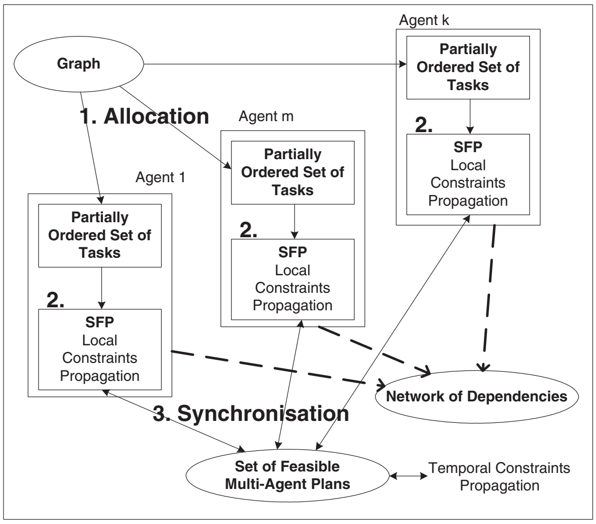

Graph
1. Allocation
Agent k
Agent m
Agent 1
Partially Ordered Set of Tasks
2.
SFP Local Constraints Propagation
Network of Dependencies
3. Synchronisation
Set of Feasible Multi-Agent Plans
Temporal Constraints Propagation

Fig. 8. The cycle for building the feasible MAP

The synchronisation of the flexible individual plans of the agents makes the MAP flexible. The SFP is also a representation of the flexibility of the agents. The more the agents own FPs in their SFP the more their plans are flexible. Intuitively speaking, during the simulation the automata in the SFPs are pruned because they no more correspond to possible states of evolution. So, at the end of the simulation, each set is reduced to only one feasible path that corresponds to the real path followed by the agents during the simulation.

6 Distributed Validation of MAP

The last step of MAP building concerns the synchronisation of the FPs. The agents have to validate the possible combinations of their FPs and the propagation of the temporal objective in these combinations. A combination is the Cartesian product of hybrid automata as defined in [8], that is to say a MAP. To validate the feasibility of the MAP, we propose an incremental algorithm of MAP validation. In fact, a combination is validated if all the agents involved in this MAP reach their final state in their respective automata and if the temporal constraints are respected. To reduce the complexity inherent to the plan flexibility, we elaborate a mechanism to keep only the possible MAPs. Indeed, even if a single agent's plan is locally feasible, it may be impossible to synchronize with the other's plans.

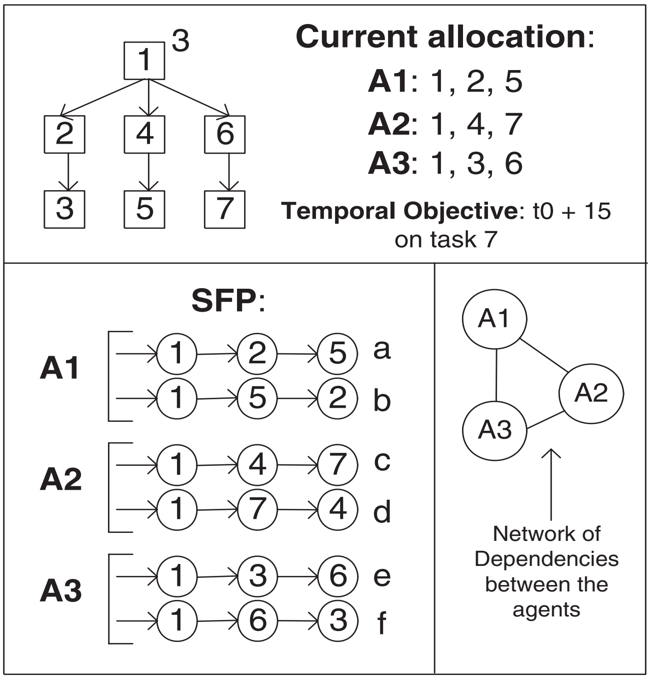

3
Current allocation:
A1: 1, 2, 5
A2: 1, 4, 7
A3: 1, 3, 6
Temporal Objective: t0 + 15 on task 7
SFP:
A1
A2
A3
a
b
c
d
e
f
A1
A2
A3
Network of Dependencies between the agents

Fig. 9. Example of SFPs

6.1 Distributed Algorithm of Validation

In figure 9, the agents have to synchronize their FPs with the others to obtain feasible MAPs. These plans must validate the temporal constraints. These constraints can be checked only during the synchronisation step because they are a shared resource of the system. The complexity comes from the possible permutations of the tasks in the planning phase. Nevertheless, some combinations are not valid and could be pruned.

To manage the synchronization, we developped a multi-agent algorithm based on the network of dependencies between the agents. Each agent tries to validate its FP taking into account the temporal constraints of the other agents. It is incremental in the way that an agent tries to validate its path till meeting a state it cannot validate. Then, it allows another agent to validate its own FP. At each step of validation, the current agent informs via the network of dependencies the concerned agent that this state is possible. Indeed, the agent unlock the others. So, at the next turn the locked agent can evolve and so on. A plan becomes feasible if all the agents have reached their final state and validate the temporal constraints. If all the agents are waiting because they are locked, the MAP is not feasible and the combination is rejected.

Algorithm of Validation

```
program ValidatePath(FeasiblePath FP, Agent Ai)
begin
  While(FP has next State to Validate)
    Q = currentState (in the automaton,
                      it corresponds to a task)
    TQ = currentTask
    If(Precondition(TQ) == true)
      state = active;
      UpDateConstraints;
      TellAgentsInRelationWith(Q, TQ, done, T);
      currentState = Next(Q, FP);
    Else
      state = WaitForPrecond(TQ);
      GiveHandOtherAgent(Aj);
      Break;
    EndIf
  EndWhile
  GiveHandOtherAgent(Aj);
end.
```

The method *UpDateConstraints* implements the temporal constraints and checks them. It can be a break point for the algorithm in the case where the temporal constraints are violated. The method $TellAgentsInRelationWith(Q, T_Q, done, T)$ informs the agents that are dependent of T_Q execution that they can evolve taking into account the interval of time T during which T_Q is done. If all the agents are in a waiting state, the plan is not feasible for functionnal reasons.

6.2 Example of Algorithm Use

In the figure 10, we have an example of the algorithm use. The first agent involved is A1.

- Step 1, it first checks the possibility to realise task 1. It is possible, so it informs all the agents of its possibility. Then it tries to validate task 2 execution. But to be executed, task 1 must have been executed three times. So A1 allows A2 to validate its path and waits for task 2 preconditions.
- Step 2, for the same reasons, A2 only validates task 1 and informs all the agents about its validation.
- Step 3, A3 can validate task 1 and task 6, it informs all the agents about task 1 validation and the agent A2 about task 6 validation (there's a link of dependency between the tasks 6 and 7).

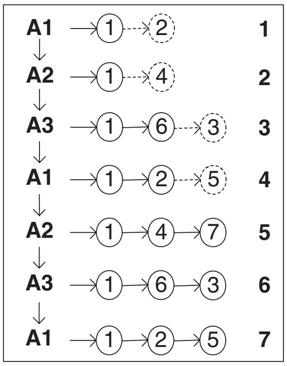


Fig. 10. Example of Algorithm Use

- Step 4, task 2 became possible, so A1 informs A3 about task 2 validation. But A1 cannot validate task 5.
- Step 5, $A2$ is now able to validate task 4. It informs A1 about it. And as 6 has been validated by A3, A2 can validate task 7 and by constraint propagation on the temporal objective, it validate the time parameter. A2 finishes its validation.
- Step 6, A3 can now validate task 3 and finishes its validation.
- Finally, step 7, A1 validates task 5 execution anf finishes its validation.

As all the agents have validated their FPs in the MAPs. All the criteria of verification are validated, so the MAP is feasible. At each step, the agent implements the temporal constraints on their automata.

6.3 Results of Validation

If we consider the SFPs presented in figure 9, the number of possible combination is $2^3 = 8$. The validation consists in checking the possible MAPs. In figure 11, we present some results of validation. In the first case (1.), the plan is feasible. The agents can reach all the states of their FPs taking into account the temporal objectives. In the second case (2.), the combination enables the agents to reach all the states of their FPs but they cannot fulfill the temporal objectives so the plan is unfeasible. In the last case (3.), the permutation engenders a break point. So, the MAP becomes simply unfeasible. In this example, the agents only keep 4 possible combinations. Moreover, it enables to reduce the SFPs of the agents. Indeed, during the synchronisation step: the FP (d) for agent $A2$ and the FP (e) for $A3$ are never used. So, it is possible to reduce their SFPs to only one FP. To sum up, this step of synchronisation enables to obtain a set of feasible MAPs, to reduce the SFPs of the agents and so the complexity of the system.

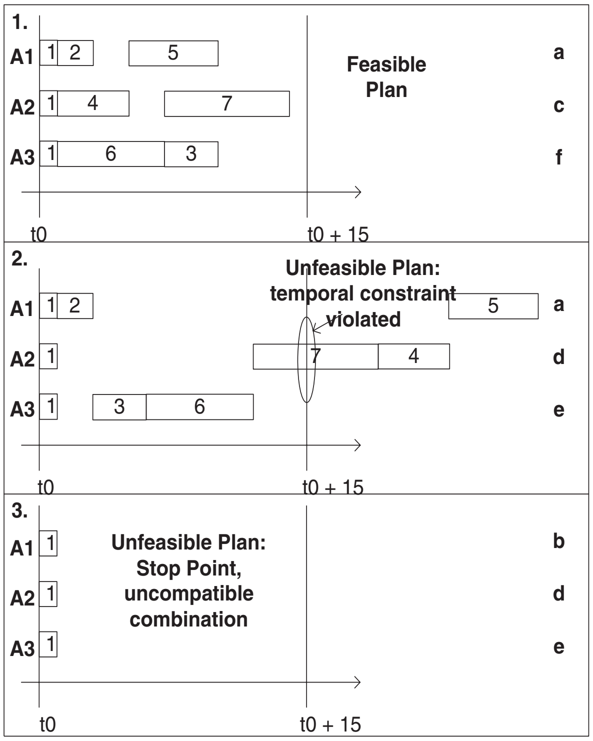


Fig. 11. Some results of validation

7 Conclusion and Future Work

In this paper, we present an approach for complex systems coordination based on multi-agent planning. From modelling by means of Hybrid Automata to the multi-agent plan validation, we show how it is possible to control all the feasible paths taking into account the flexibility of the agents in this process. Our future work aims to extend our approach with operators on the plans (e.g. merging, concatenation, insertion, etc.) to deal with dynamic reallocation of tasks.

References

1. Martial, F.V.: Coordinating Plans of Autonomous Agents. Springer-Verlag, LNAI (1992)
2. Alami, R., Robert, F., Ingrand, F., Suzuku, S.: A paradigm for plan-merging and its use for multi-robot cooperation. In: Proceedings of IEEE International Conference on Systems, Man and Cybernetics, San Antonio, Texas (1994)
3. Durfee, E.H.: Distributed problem solving and planning. Multiagent Systems A Modern Approach to Distributed Artificial Intelligence **1** (1999) 121–164
4. ElFallah-Seghrouchni, A., Haddad, S.: A coordination algorithm for multi-agent planning. In: Proceedings of MAAMAW'96, LNAI 1038, Springer-Verlag (1996)

5. ElFallah-Seghrouchni, A., Haddad, S.: A recursive model for distributed planning. In: Proceedings of ICMAS'96, AAAI Press (1996)
6. ElFallah-Seghrouchni, A., Degirmenciyan-Cartault, I., Marc, F.: Framework for multi-agent planning based on hybrid automata. In: LNAI 2691, CEEMAS 03 (2003) 226–235
7. Marc, F., ElFallah-Seghrouchni, A., Degirmenciyan-Cartault, I.: Multi-agent planning as a coordination model for self-organized systems. In: IEEE, IAT 03 (2003)
8. Henzinger, T.A.: The theory of hybrid automata. In: Proceedings 11th IEEE Symposium Logic in Computer Science, LCAIS'96 (1996) 278–292
9. Musliner, D.J.: CIRCA: The Cooperative Intelligent Real-Time Control Architecture. PhD Thesis, University of Michigan (1993)
10. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science **126** (1994) 183–235
11. Goldman, R.P., Musliner, D.J., Pelican, M.J.S.: Exploiting implicit representations in timed automaton verification for controller synthesis. In: Proceedings 2002 Hybrid Systems: Computation and Control Workshop. (2002)
12. Howden, N., Ronnquist, R., Hodgson, A., Lucas, A.: Jack intelligent agents summary of an agent infrastructure. 5th Conference on Autonomous Agent (2001)
13. Henzinger, T.A., Ho, P.H., Wong-Toi, H.: Hytech : a model checker for hybrid systems. Journal of Software Tools for Technology Transfer **1** (2001) 110–122
14. Raskin, J.F., Schobbens, P.Y.: The logic of event clocks, decidability, complexity and expressiveness. Journal of Automata, Languages and Combinatorics **4** (1999)
15. Degirmenciyan-Cartault, I., Marc, F., ElFallah-Seghrouchni, A.: Modeling multi-agent plans with hybrid automata. In: Proceedings of the Formal Approach for Multi-Agent Systems Workshop (FAMAS 03), ETAPS 03 (2003)
16. ElFallah-Seghrouchni, A., Marc, F., Degirmenciyan-Cartault, I.: Modelling, control and validation of multi-agent plans in highly dynamic context. In: ACM. To be published, AAMAS 04 (2004)

Author Index