AAMAS 2010
TORONTO

The 9th International Conference on
Autonomous Agents and Multiagent Systems
May 10-14, 2010
Toronto, Canada

Workshop 27

The Eighth International Workshop on Programming Multi-Agent Systems

ProMAS 2010

Editors:
Wiebe van der Hoek
Gal A. Kaminka
Yves Lespérance
Michael Luck
Sandip Sen

The Eighth International Workshop on Programming Multi-Agent Systems

11th May 2010, Toronto, Canada

Rem Collier
(University College Dublin)

Jürgen Dix
(Clausthal University of Technology)

Peter Novák
(Czech Technical University in Prague)

Preface

These are the proceedings of the International Workshop on Programming Multi-Agent Systems (ProMAS2010), the eighth of a series of workshops that is aimed at discussing and providing an overview of the current state-of-the-art technology for programming multi-agent systems.

The aim of the ProMAS workshop series is to promote research on programming technologies and tools that can effectively contribute to the development and deployment of multi-agent systems, with a particular focus on promoting the discussion and exchange of ideas concerning the techniques, concepts, and principles that are important for establishing multi-agent programming platform that are useful in practice and have a theoretically sound basis. Topics typically addressed include: the theory and application of agent programming languages; the verification and analysis of agent systems; and the implementation of social scaffolding to support organisation, coordination and communication within agent-based systems.

In its previous editions, which took place during AAMAS 2003 (Melbourne, Australia), AAMAS2004 (New York, USA), AAMAS 2005 (Utrech, Netherlands), AAMAS 2006 (Hakodate, Japan), AAMAS 2007 (Honolulu, Hawaii), AAMAS 2008 (Estoril, Portugal), and AAMAS 2009 (Budapest, Hungary), ProMAS constituted an invaluable occasion to bring together leading researchers from both academia and industry to discuss issues on the design of programming languages and tools for multi-agent systems. We are very please to be able to continue this tradition with this years edition, to be held on May 11th as part of AAMAS 2010, which is being held in Toronto, Canada, and have accepted 6 papers for presentation.

At the workshop, in addition to the regular papers, two invited talks will be given. The first, by Gregory O'Hare (University College Dublin, Ireland), focuses on the issue of programming intelligent agents for Ubiquitous Sensing Devices. Ubiquitous Systems, and in particular, Wireless Sensor Networks (WSN) represent an important potential application domain for multi-agent systems due to their high levels of distribution, and their ad-hoc nature. The sheer scale of the expected deployments raises a number of significant number of challenges both in terms of managing the well-being of the network, and also in terms of mediating the demands of heterogeneous applications that are competing for use of a shared set of sensor resources. From a programming multi-agent systems perspective, this requires the availability of agent technologies that: offer minimized footprints for BDI style agents; effective support for agent mobility; and utility-based resource aware reasoning. In his talk, O'Hare outlines recent work on addressing these challenges via the use of Agent Factory Micro Edition (AFME) a shrink wrapped BDI software framework, that has been utilized with the latest genre of sensor and its use within SIXTH, a WSN middleware that incorporates an agent based programming metaphor which supports effective (re)programming of WSNs.

The second talk, by Sarit Kraus (University of Maryland, USA/Bar Ilan University, Israel) focuses on the issue of Human-Computer Negotiation. Negotiation

is a core issue in multi-agent systems research. While it has been well studied in terms of agent-agent interaction, the continuing trend towards the dissemination of applications such as the Internet across geographical and ethnic borders has opened up opportunities for computer agents to negotiate with people of diverse cultural and organizational affiliation. Designing automated negotiators that are able to proficiently interact and collaborate with their human partners is a significant challenge as people often follow suboptimal decision strategies due to irrationalities attributed to lack of knowledge of own preferences, the effects of task complexity, framing effects, the interplay between emotion and cognition, and the problem of self control. In her talk, Kraus describes a new agent for bilateral negotiation in repeated interactions, known as PURB, that is specially designed to adapt to the particular behavioural traits of its negotiation partner and outlines the results of an extensive study that evaluated the performance of PURB when interacting with people in three different cultures.

As in previous editions, the themes addressed in the accepted papers included in this Volume range from technical topics related to, for example, agent-environment interaction to conceptual issues related to, for example, reasoning about partial goals or action-rule preference heuristics.

Agents and Environments/Organisations

The paper by Carr et al., describes a new multi-agent programming environment, known as PreSage-MS, which provides a rapid prototyping and animation tool that is designed to facilitate experiments with organised adaptation in teams of agents that are imbued with complex reasoning abilities. The goal of the work is to provide tool support that would allow multi-agent system developers to develop agents that are based on dynamic protocol specifications and to evaluate how teams of agents adapt via a simulation environment.

The paper by Behrens et al., explores the interface between agents and their environments and in particular, the lack of consistency that currently pervades the development of environments and the impact of that lack of consistency on the complexity of integrating disparate agent systems with an environment. The main focus of the work is to propose a standard for the interface between an agent and its environment that is based on previous experiences gained from integrating heterogeneous agent systems as part of the annual Agent Contest. The proposed standard is evaluated through a reference implementation (the Agent Contest Server) and its integration with four well-know agent programming languages: 2APL, GOAL, Jadex, and Jason. The same integration was then used with two environment implementations: the Agent Contest cow herding scenario, and an elevator environment.

The paper by Ricci et al. discusses the limitations of the traditional model of perception and action utilized in agent programming languages when applied to endogenous environments. Further, in light of these limitations, the paper proposes a new model of perception and action that is better suited to such environments. The model is then evaluated through a combination of CArtAgO as

the environment technology and three reference agent programming languages: Jason, 2APL, and GOAL.

Agent Programming

The paper by Jordan and Collier describes early work that aims to investigate multi-paradigm metrics that can be used to evaluate agent-oriented programs. Metrics are an important tool in a modern software engineers armory as they can be highlight software defects or identify code that will be difficult to maintain. Multi-paradigm metrics aim to measure the structural complexity of software that has been implemented using more than one programming paradigm. The paper proposes a basic meta model and process for evaluating cohesion and coupling within a programming language. The approach is applied to a simple Jason program.

Agent Reasoning

The paper by van Riemsdijk and Yorke-Smith outlines proposes a new model for representing goals that allows for partial goal satisfaction with respect to achievement goals. The work is motivated by an example problem involving an accident at a chemical plant and is formalised by a higher-level framework based on metric functions that support the representation of concepts such as progress towards the achievement of a goal.

The paper by Broekens et al., presents an approach to prioritising action-rule selection based on reinforcement learning. The main objective of the work is to overcome the problem of underspecified agent programs by introducing a reinforcement learning component that allows that agent to prioritise action-rules based on experience. The approach adopted employs a domain independent heuristic that offers a significant improvement in the behaviour of the agent. This is demonstrated through an experimental analysis based on blocks world.

March 2010

Rem Collier
Jürgen Dix
Peter Novák

Conference Organization

Programme Chairs

Rem Collier Jürgen Dix Peter Novák

Programme Committee

Matteo Baldoni Guido Boella Juan Botia Lars Braubach Louise Dennis Ian Dickinson Mauro Dragone Berndt Farwer Michael Fisher Jorge Gomez-Sanz Vladimir Gorodetsky James Harland Koen Hindriks Benjamin Hirsch Jomi Fred Hubner Joao Leite Viviana Mascardi John-Jules Meyer Joerg Mueller Andrea Omicini Frederic Peschanski Michele Piunti Agostino Poggi Alexander Pokahr Alessandro Ricci Ralph Ronnquist Sebastian Sardina Ichiro Satoh Munindar P. Singh Tran Cao Son Kostas Stathis Paolo Torroni Gerhard Weiss Wayne Wobcke Neil Yorke-Smith Yingqian Zhang boissier olivier M. Birna van Riemsdijk Leon van der Torre

External Reviewers

Stefano Bromuri

Table of Contents

Invited talks.

Programming Agents for Ubiquitous Sensing Devices *(invited talk)* 1
Gregory O'Hare

Human-Computer Negotiation: Learning from Different Cultures *(invited talk)* 3
Sarit Kraus

Session 1. Agents and Environments/Organisations

Software Support for Organised Adaptation 5
Hugo Carr, Alexander Artikis, Jeremy Pitt

Action and Perception in Multi-Agent Programming Languages: From Exogenous to Endogenous Environments 21
Alessandro Ricci, Andrea Santi, Michele Piunti

An Interface for Agent-Environment Interaction 37
Tristan Behrens, Jürgen Dix, Koen Hindriks, Mehdi Dastani, Rafael Bordini, Jomi Hübner, Alexander Pokahr, Lars Braubach

Session 2. Agent Programming

Evaluating Agent-Oriented Programs: Towards Multi-Paradigm Metrics . . 53
Howell Jordan, Rem Collier

Session 3. Agent Reasoning

Towards Reasoning with Partial Goal Satisfaction in Intelligent Agents . . 69
M. Birna van Riemsdijk, Neil Yorke-Smith

Reinforcement Learning as Heuristic for Action-Rule Preferences 85
Joost Broekens, Koen Hindriks, Pascal Wiggers

Programming Agents for Ubiquitous Sensing Devices

G. M. P. O'Hare

CLARITY: Centre for Sensor Web Technologies
School of Computer Science and Informatics, University College Dublin (UCD), Ireland
gregory.ohare@ucd.ie

Abstract

Ubiquitous sensing envisages a world which is saturated with sensing devices, a world in which traditional boundaries between the digital world, and the physical world which we inhabit, will be systematically deconstructed producing a data reservoir which is seamless, vast, streamed, heterogeneous, noisy, incomplete and contradictory. Such ubiquitous sensing devices (motes) are typified by their resource-bounded nature. They are computationally challenged in terms of processing capabilities, memory and power, the later due to their battery powered operation. Motes afford three key roles those of sensing, actuation and routing.

By virtue of wireless communications individual sensing devices can function in concert with a typology of sensors forming a Wireless Sensor Network (WSN). It is recognized that decisions taken by an individual sensing node necessarily have implications for neighbouring sensors. Consequently decisions by an individual sensor to, for example degrade sampling frequency or increase hibernate cycle, ought to be taken in a collaborative manner with fellow adjoining sensors. The deliberation associated with such behavioural changes together with the collaborative nature of the decision making have suggested the appropriateness of a Multi-Agent Systems (MAS) approach [1], [2]. Numerous research have already explored the use of reactive style agents in WSNs for example Agilla [3] while the efficacy of deliberative, specifically Belief Desire Intention (BDI), have been demonstrated in network power management [4].

The challenge of (re)programming such sensor devices should however not be underestimated. This new generation of pervasive computing device represents a new frontier for the programming of multi-agent systems and, one which presents a number of key challenges. These include:

- Minimizing the software footprint for BDI style agents;
- Effective support for agent mobility;
- Utility-based resource-aware reasoning;
- A middleware for supporting (re)programming of WSN;

This paper addresses these challenges and demonstrates efforts that have been advanced in each of these areas. In particular it utilizes Agent Factory Micro Edition (AFME) [5] a shrink wrapped BDI software framework, which has been utilized with the latest genre of sensor. This new generation of sensor platform is more powerful typically offering a powerful RISC processor together with a Java Virtual Machine

(JVM) like Squawk. Sunspot or Stargate typify such devices. In such java based environments clearly only weak migration may be supported. However reliable and robust migration must be supported which like other agent activity is ever aware of the cost of deliberation in terms of available battery power. To this ends it is necessary to introduce regimes which take due cognizance of computation cost and thus provide utility-based resource-aware reasoning [6].

Finally this paper introduces SIXTH a WSN middleware that incorporates an agent based programming metaphor which supports effective (re)programming of WSNs.

Keywords

Ubiquitous Sensing, Multi-Agent Systems (MAS), Embedded Systems, Wireless Sensor Networks, Agent Factory, Wireless Sensor Network Middleware.

Acknowledgements

Gregory O'Hare gratefully acknowledges the support of Science Foundation Ireland under Grant No. 03/IN.3/1361.

References

[1] Marsh, D., Tynan, R., O'Kane, D. & O'Hare, G.M.P., Autonomic Wireless Sensor Networks, Engineering Applications of Artificial Intelligence Journal, Vol 17. No. 7, pp 741-748, Elsevier Science, 2004.

[2] Tynan, R. Muldoon, C., O'Hare, G.M.P. & O'Grady, M.J., Coordinated Intelligent Power Management and the Heterogeneous Sensing Coverage Problem, The Computer Journal (Special Issue on Agent Technologies for the Sensor Networks), Oxford University Press, 2010 (In Press).

[3] Fok, C., Roman, G., and Lu, C. 2009. Agilla: A mobile agent middleware for self-adaptive wireless sensor networks. ACM Trans. Auton. Adapt. Syst. 4, 3 (Jul. 2009), pp 1-26.

[4] David Marsh , Donal O'Kane , G. M. P. O'Hare, Agents for Wireless Sensor Network Power Management, Proceedings of the 2005 International Conference on Parallel Processing Workshops, p.413-418, June 14-17, 2005.

[5] Muldoon, C., O'Hare, G.M.P. & O'Grady, M.J., AFME: An Agent Platform for Resource Constrained Devices, In Proceedings of 7th International Workshop, Engineering Societies in the Agents World VII, ESAW 2006. O'Hare, G.M.P., Ricci, A., O'Grady, M.J. & Dikenelli, O. (Eds.), University College Dublin, Dublin, Ireland 6th- 8th Sept. 2006.

[6] Shen, S, O'Hare, G.M.P., O'Grady, M.J. Fuzzy Decision Making through Energy-aware and Utility Agents within Wireless Sensor Networks. Artificial Intelligence Review, Vol. 27, No. 2/3, Springer, 2007.

Human-Computer Negotiation: Learning from Different Cultures*

Sarit Kraus

Dept. of Computer Science
Bar-Ilan University Ramat Gan 52900 Israel and

Institute for Advanced Computer Studies
University of Maryland, College Park MD 20742 USA

Abstract. Negotiation is a process by which interested parties confer with the aim of reaching agreements. The ability to negotiate successfully is critical for many social interactions, The dissemination of applications such as the Internet across geographical and ethnic borders are opening up opportunities for computer agents to negotiate with people of diverse cultural and organizational affiliation. These automated negotiators should be able to proficiently interact and collaborate with their human partners. However, people often follow suboptimal decision strategies due to irrationalities attributed to lack of knowledge of own preferences, the effects of the task complexity, framing effects, the interplay between emotion and cognition, and the problem of self control. Furthermore, culture plays an important role in their decision making and people of varying cultures differ in the way make offers and fulfil their commitments in negotiation. In this talk we will describe a new agent for bilateral negotiation in repeated interactions that allow players to renege on agreements. PURB was especially designed to adapt to the particular behavioural traits of its negotiation partner. Its strategy is composed of a utility function that depends on the extent to which the other player is reliable and helpful, as well as rule-based mechanism that uses this utility for generating and replying to offers, and for deciding whether to fulfil its agreements. I will present an extensive study that evaluated the performance of PURB when interacting with people in three different cultures. Our results show that the performance of the PURB agent directly depended on the cultural affiliation of its negotiation partners: People's negotiation behaviour, in particular the extent to which they fulfil agreements, varies widely across cultures, and this had a crucial effect on PURB's performance. I will also present additional results that compare the performance of PURB when playing Peer-Designed-Agents — agents that were designed by non-experts to represent themselves during negotiation. I will conclude by demonstrating how the PURB agent could be improved using the collected data.

* This research is a joint work with Ya'akov Gal and Michele Gelfand. It is based upon work supported in part by the U.S. Army Research Laboratory and the U.S. Army Re- search Office under grant number W911NF-08-1-0144 and under NSF grant 0705587.

Software Support for Organised Adaptation

Hugo Carr[1], Alexander Artikis[21], Jeremy Pitt[1]

[1] Electrical & Electronic Engineering Department,
Imperial College London, SW7 2BT
[2] Institute of Informatics and Telecommunications,
National Centre for Scientific Research "Demokritos", Athens 15310

Abstract. Coordination of agent teams depends on the scale of the team. Teams comprising hundreds of agents tend to perform better with local computation and interactions (i.e. swarm intelligence, evolutionary computing, etc.) Teams comprising tens of agents tend to perform better with more sophisticated agents (e.g. BDI agents) with complex reasoning abilities. In the long term, our objective is to achieve agent teams comprising hundreds of sophisticated agents: then, a key aspect of coordination and control is the idea of organised adaptation. In this paper we present a new multi-agent programming environment, **Pre**Sage-$\mathcal{MS}$, a rapid prototyping and animation tool designed to facilitate experiments with organised adaptation in 'sophisticated' agent teams. We describe the system architecture and functionality, and give a walkthrough of experimental design. We conclude with a discussion of several issues, including the migration from design-time tools for human users to run-time services for software agents.

Categories and subject descriptors: I.2.5 [**Artificial Intelligence**]: Programming Languages and Software; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Intelligent Agents*

General terms: Design, Experimentation

Keywords: Agent-oriented programming, organised adaptation, dynamic specification, temporal logic

1 Introduction

Coordination of agent teams depends on the scale of the team. To date, teams comprising hundreds of agents tend to perform better with local computations and interactions, with each other (swarm intelligence [12]) or with their physical environment (stigmergy [10]). Teams comprising tens of agents tend to perform better with more sophisticated agents with complex reasoning abilities (e.g. BDI agents [11]). In the long term, our ultimate objective is to deploy agent teams comprising hundreds of sophisticated agents: then, a key aspect of coordination and control will then be the idea of *organised adaptation*.

Organised adaptation, as opposed to emergent behaviour, is the conscious, deliberate and targeted adaptation of a specification and/or configuration of a

multi-agent system, in response to systemic requirements or environmental conditions. Emergent behaviour produces unintended or unknown global outcomes derived from hard-wired local computations, with respect to the environment and/or physical rules. Instead, we are concerned with the introspective application of soft-wired local computations, with respect to the environment, physical rules and conventional rules (what some philosophers of language would call 'constitutive rules'), to produce intended and coordinated global outcomes.

The complexity of modern software systems demands that organised adaptation be an on-line mechanism, that proceeds without human intervention or supervision (cf. autonomic computing [13]); in other words, to make the mechanics for organised adaptation available to software agents at run-time. In this paper we present a new multi-agent programming environment, **Pre**Sage-$\mathcal{MS}$, which converges the multi-agent programming environment **Pre**Sage [15] with the analytic tool of [1]. The result is a rapid prototyping and animation tool designed to facilitate experiments with organised adaptation of dynamic specifications at run-time, for 'sophisticated' agents (i.e. agents with complex reasoning capability wrt. adaptation, goals, other agents, etc.), operating (eventually) in 'large' teams.

In this paper, we present the **Pre**Sage-$\mathcal{MS}$ system architecture and functionality, and give a walkthrough of system design. Accordingly, this paper is organised as follows. The next section outlines how a dynamic specification can be defined, and pre-existing tools for evaluating such specifications. Section 3 describes the architecture of, and functionality supported by, the new system. A walkthrough of experimental design in **Pre**Sage-$\mathcal{MS}$ is given in Section 4. We conclude with a discussion of the current, related and future work; in particular we discuss the role of institutional agents in the migration from design-time tools for human users to run-time services for software agents.

2 Background Work

In this section, we present the background to the current work. First we discuss dynamic protocol specifications, then we present two existing software tools used for experimenting with different aspects of dynamic protocols. This is the basis for the system description of **Pre**Sage-$\mathcal{MS}$ in Section 3, which is a convergence and enhancement of both these tools.

2.1 Dynamic Protocol Specification

Consider the two following examples:

Example 1: Resource-sharing protocol: There is a set of agents S, a subset of which occupies the role of *subjects* who are entitled to access a resource, and a designated agent in S occupying the role of *chair*. The subjects are empowered to request access to the resource, the chair is empowered to grant or revoke access. The protocol stipulates that one or more subjects request access to the

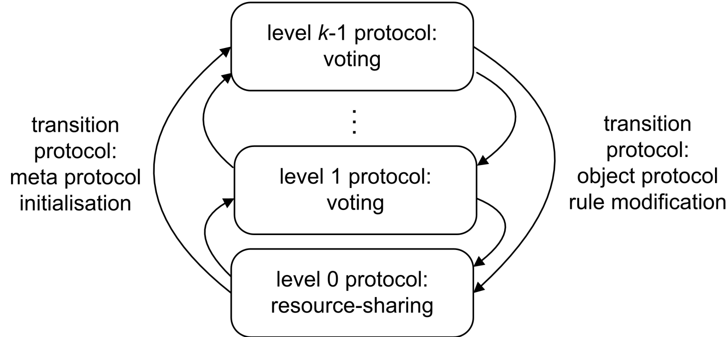

Fig. 1. A k-level Infrastructure for Dynamic Specifications.

resource, the chair grants access to only one; that agent uses the resource until it is released it or access is revoked, and the cycle repeats.

Example 2: Voting protocol: There is a set of agents S, a subset of which occupy the role of *voters* who are entitled to vote, and a designated agent in S occupying the role of *returning officer*, who declares the result of a vote. The protocol stipulates that the officer calls for a ballot on a specific motion, the voters cast their votes (express their preference), the officer counts the votes and declares the result according to the standing rules.

In both examples, there are values which may be changed, even during execution of the protocol. In the first example, which agent occupies which role or roles, the rule by which access is granted, the period until the chair has permission to revoke access, etc., may all be changed. Similarly, in the second example, role assignment again is mutable, and there are many parameters to a vote: single or multiple winner, standing rules for winner determination (plurality, run-off, borda count, etc.), votes required to be quorate, and so on.

One key aspect of organised adaptation are dynamic specifications, which allow agents to alter the rules of a protocol P, even during the protocol execution. P is considered an 'object' protocol, if at any point in time the participants may start a 'meta' protocol in order to decide whether the object protocol rules should be modified to P' (say). Moreover, the participants of the meta-protocol may initiate a meta-meta-protocol to decide whether to modify the rules of the meta-protocol, and so on. Figure 1 shows an infrastructure for dynamic resource-sharing protocols, that is, the object protocol is a resource-sharing protocol, and every $\{\text{meta}^+\}$-protocol is (some type of) a voting protocol.

Apart from object and meta protocols, the infrastructure for dynamic specifications includes 'transition' protocols (again, see Figure 1) that is, procedures that express, among other things, the conditions under which an agent may validly initiate a meta-protocol, the roles that each meta-protocol participant will occupy, and the ways in which an object protocol is modified as a result of the meta-protocol interactions.

2.2 System Support for Adaptive Specifications

Adaptive Behaviour. **Pre**Sage is a simulation platform for agent animation and rapid prototyping of societies of agents. It offers a multi agent systems programmer a flexible and generic set of Java classes, interfaces and tools with which key aspects of agent societies can be designed and simulated.

To develop a prototype in **Pre**Sage, it is necessary to define agent participant types: this can be done by extending the abstract class supplied with standard environment (to guarantee compatibility with the simulation calls and provide core functionality like message handling etc.) or by defining a new class. The **Pre**Sage environment is in fact neutral with respect to the internal architecture of its agents: thus agents can be of arbitrary complexity. Agents can then either be animated (i.e. a 'sophisticated' agent with complex reasoning capability is actually embedded in an artificial environment) or simulated (complex behaviour is approximated by simulation rather than actually computed).

Then the network properties and physical world are defined, using or extending the given base classes. Finally, additional plugins can be written for visualisation, connection to other components (e.g. a database for logging results, etc.), or generation of exogeneous events.

We have used **Pre**Sage for initial experiments in organised adaptation. We set up a simple iterated 'tragedy of the commons' scenario, with partial knowledge, no central control, and self-interested agents, and allowed the agents two votes: one for whom to allocate resources, and one to decide how many votes should be received in order to be allocated resources. The idea was that co-operative agents should manage the system by voting 'fairly'. Initial experiments showed that 'responsible' agents performed better than selfish or cautious ones (indeed approximated the outcomes achieved with a 'benevolent dictator') [5], and that social networking (gossiping) algorithms can be used on an individual and group basis to protect the system from self-interested behaviour [6].

Metric Space Analysis. Given an adaptable specification, the protocol rules and parameters that may be modified at run-time are called *Degrees of Freedom* (*DoF*). Each DoF can take one value from a specific set of possible values; we map each of the possible values onto a rank order. A specification with m DoFs can then be represented as an m-tuple, where each tuple element defines the rank order of the value for the corresponding DoF. The set S of all possible tuples is given by all the possible instantiations of every DoF with the rank order of each of its possible values. Clearly there are:

$$|v_1| \times |v_2| \times \ldots \times |v_m|$$

members of this set, where $|v_i|$ is the maximum rank value of the ith DoF $v_i (1 \leq i \leq m)$ can take. This set is the basis of a *metric space* $\mathcal{M} =< S, d >$ if we define a metric d on the set which defines a 'distance' between any pair of set members (subject to the usual constraints [16]).

We can use this representation of an m-dimensional specification as a metric space by measuring the 'distance' between members of the set, or rather, *specification points* in the metric space. A designer can then define an adaptable specification with its degrees of freedom, and additional constraints on run-time modification. For example, the designer could specify a 'desired' specification point, and proposed modifications could be evaluated on the 'distances' of the proposed specification point from the 'current' point and the 'desired point. The designer could also specify that some points are 'forbidden', if for example they were normatively inconsistent [2].

The metric space representation was the basis of automated support for design-time analysis of a dynamic protocol specification, that is, an off-line, static analysis of a protocol that could be adapted at run-time [16]. This tool allowed the designer to analyse a narrative of events (actions taken by agents to modify the specification) and determine, at each time point, the distance to the desired specification point for a range of different metrics (euclidean, manhattan, weighted manhattan, etc.) This allowed the designer to evaluate comparatively the effects of different metrics on different instances of a dynamic specification.

3 PreSage-$\mathcal{MS}$

Both of the tools described in the previous section are useful for investigating certain aspects of organised adaptation but are ultimately limited. PreSage allowed mixed agent strategies and populations, but the agents did not have an explicit representation of metric spaces; while the second tool is restricted to a retrospective, design-time analysis of a given narrative of events. Ultimately, we want to perform an introspective, run-time analysis *as the narrative unfolds.* Therefore, we have developed a new system, **Pre**Sage-$\mathcal{MS}$, which retains and integrates the agent-level granularity of **Pre**Sage with the metric space analysis of [2], but extends both by equipping the agents themselves with the functionality to represent and reason about metric spaces and specification points.

In this section, we present the architecture and functionality of the **Pre**Sage-$\mathcal{MS}$ programming environment.

3.1 PreSage-$\mathcal{MS}$ Architecture

The core **Pre**Sage System is conceptually composed of three layers: the base simulation layer, the services layer, and the instances layer.

The base simulation layer performs parameter initialisation, manages the simulation execution, and provides generic functions to higher level modules and classes. **Pre**Sage uses a multi-agent discrete-time-driven simulation model. In this model, each loop of the simulator control thread equates to a time-slice of the simulated multi-agent system. In each time-slice, the agent participants are given a turn to perform physical and communicative actions, the state of the network(s) and physical world is updated, scripted events are executed, and plugins perform their specified operations.

The services layer provides the skeleton models that designers use to implement a simulation. The agents and their environment are implemented through the provided interfaces, simulating network(s), and a physical world. The managers handle scripted events and other user plugins as well as providing a dedicated Event Calculus tool to improve latency in rule unification.

The third layer comprises the user-specified instances of these components.

PreSage-$\mathcal{MS}$ extends **Pre**Sage by implementing the Event Calculus (EC) [14] as a universal language for communication and specification. Using the EC, the agents can establish a narrative of speech acts, whose consequences can be calculated as a normative state; ie. the set of permissions, powers and obligations of agents according to which roles they occupy. These norms can be derived by unifying the narrative with the object and meta-level rules can also be written in the EC.

To integrate the EC with **Pre**Sage, we have implemented a new manager at the interface level which handles the EC requests and keeps track of the EC fluents. A fluent is a value which varies over time, so the EC manager keeps track of its current value to prevent unnecessary queries to the temporal calculus engine. This manager uses JPL, an interface to Prolog from Java, by keeping a Prolog implementation of the EC and the implementation specific rules.

At the instance level, we have included

- An environment in which agents may navigate a centrally managed metric space
- An extendable agent which can send and interpret EC messages
- An EC plugin which displays the object and meta level protocol read by the EC manager, and the fluents which 'hold' at the current time point.
- A metric space plugin which allows a system designer to rank the DoF values and alter the metric space of the system

Both plugins act as run-time services for participants, providing live information about the EC fluents and metric space.

The final architecture of **Pre**Sage-$\mathcal{MS}$ is illustrated in Figure 2. Next, we will look at the functionality of the **Pre**Sage extensions in more detail.

3.2 Agents and Environment

Simulation design in **Pre**Sage-$\mathcal{MS}$ has been divided into the environment, and the agents which act therein. At its simplest, an environment may act only as a communication link between the agents, but it is often convenient to keep a central representation of the object and meta-level rules. For example, when designing systems which have dynamic specifications, it is usually desirable to include the system's specification space in the environment. This serves as a central reference for newly registered agents, and ensures that there is no confusion about where in the metric space the system lies.

Environments in **Pre**Sage-$\mathcal{MS}$ go further than the basic message passing paradigm and handle actions made by agents by broadcasting the appropriate

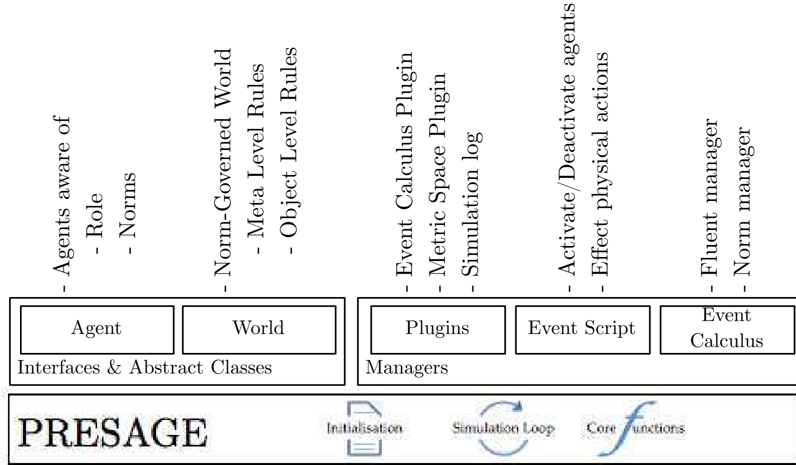

Fig. 2. Architecture of the **Pre**Sage-$\mathcal{MS}$ system

speech act to the agents affected. These speech acts are similarly sent to the Event Calculus manager to update the narrative (ie. the action history). The EC Manager maintains the metric space and object level rules in Prolog, and with a minimal number of queries, sends back the new set of norms effected by the speech act. In order for agents in **Pre**Sage-$\mathcal{MS}$ can understand messages from the environment, we have tools for parsing event calculus messages and a set of fluent and norm handlers which are invoked when an update message relating to a fluent or norm is received from the environment.

3.3 Event Calculus Manager

The event calculus manager acts as a buffer between **Pre**Sage-$\mathcal{MS}$ and the declarative implementation of the Event Calculus (Figure 3). The EC derives the norms at a timepoint T by unifying the history of speech acts with the predicates describing the rules of the system. However, as the action history increases the computation time becomes prohibitively slow. The EC manager has therefore been optimised to reduce the number of queries to the Prolog knowledgebase by implementing a caching mechanism to bound the size of the action history.

We have included an interface to the event calculus manager, which monitors the current states of the event calculus fluents and norms. This front end receives updates from the manager with respect to speech acts and their consequences. Agents who for whatever reason are not able to receive messages from the environment, can register with this service to check that their local representation of system norms is consistent. Designers may also use this inspector to view the history of all fluent and norm changes throughout the life cycle of the system.

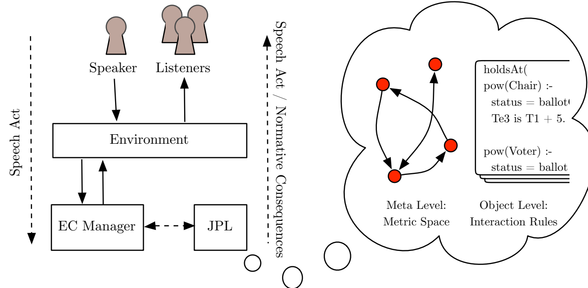

Fig. 3. The relationship between the Agents of a system, the environment and the event calculus specification of the object and meta-level rules. Calls to the Prolog implementation of the event calculus, through the JPL library, are minimised by the EC Manager.

3.4 Metric Space Plugin

The metric space plugin in Figure 4 represents an extension of the analytic tool in [1]. The plugin implements much of the same functionality as the original, but includes a graphical visualisation of the metric space based on a selected metric; run-time services to supply participants metric space specific information in real time; and a generalisation of the ranking process for DoF values.

As demonstrated in section 2.2, DoFs can be ranked using a one dimensional number for each of the DoF values. We have generalised this model by ranking each value with a more general fixed size vector.

In a specification of m DoFs, each DoF still takes one value from specific set of possible values but we map each of the possible values onto the domain $\mathbb{R}^{n_k}$ where $0 \leq k < m$. Rank order no longer exists, as the relationship between the DoF values becomes more sophisticated and vectors cannot be compared in terms of higher versus lower. However, a metric space is still formed provided that the metrics used within the vectors are the same as those for the space which they form.

The metric space visualisation draws a graph of specification points based on a selected metric (Euclidean, Manhattan etc.) and threshold distance. This threshold represents the furthest distance value that an edge can take between two specification points, and may be used to represent several things. For example, it may relate to the furthest distance that an agent may travel away from the current point. This may be due to the cost of adaptation being too high, or an attempt to limit the volatility of the system by making incremental changes. This results in an agent having to choose the shortest path along the edges of the graph to reach a desired specification point.

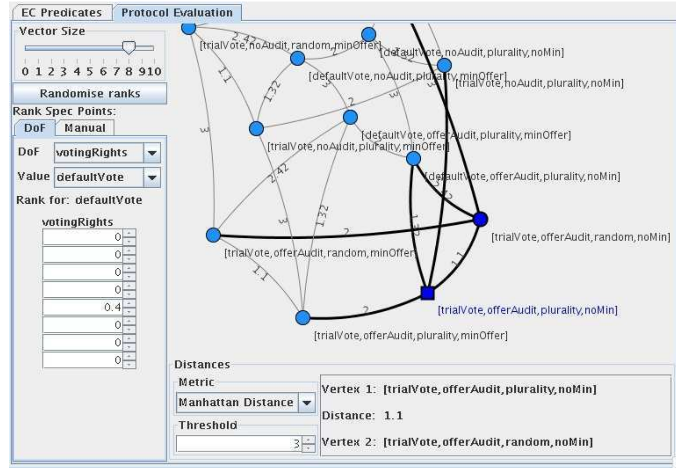

Fig. 4. Interface to the metric space: The degrees of freedom of the metric space are resized and ranked in the lefthand column. The bottom panel selects which metric the system is using to form the space, and the graph is drawn according to the threshold value which represents the maximum value that an edge may take.

4 PreSage Experimental Design

System designers operate **Pre**Sage-$\mathcal{MS}$ in three inter-related stages: Agent design, protocol design and metric space design. On completion of these stages, experiments exploring the metric space can be designed.

A system lifecycle begins offline with the Java and Prolog implementation of the agents and protocol. The designer goes on to choose how the agents are to reason about the DoFs by formulating a metric space which represents a set of measurements agents may use to compare different specifications. The metric can be defined in conjunction with further permissions, powers and obligations which constrain how the system may adapt.

To demonstrate the experimental design in **Pre**Sage-$\mathcal{MS}$ we have provided a walkthrough of the resource allocation example outlined in section 2 with four object level DoFs and a static meta level protocol. Based on this architecture we outline a set of experiments which we intend to address investigating *when* agents should adapt.

4.1 Agent Design

The agent design process results in a test population which recreates the conditions of an open system. Agents, along with their EC handlers, are designed offline in Java with extendable **Pre**Sage-$\mathcal{MS}$ interfaces. They must then be endowed with the capacities required to approximate independent decision making.

At the most basic level agents require a complete set of norm and fluent handlers for the parts of the protocol which they are involved in. For example in the resource allocation scenario we have a set of basic speech acts which all agents must be able to interpret (openSession, callForProposals, propose). A propose handler must be able to read the proposal made by an agent and store the offer and request for use during the voting protocol:

```
protected class ProposeHappensHandler extends HappensHandler {
        public void handle(ProposeHappens happens) {
                String[] arguments = happens.getArguments();
                requests.put(arguments[0], new Integer(arguments[1]));
                offers.put(arguments[0], new Integer(arguments[2]));
                totalRequests+=Integer.parseInt(arguments[1]);
        }
}
```

Agents require the means to reason about the dynamic protocol, this may require specific knowledge about the protocol. There are however, general ways of navigating a specification space which can be used in conjunction with machine learning techniques and heuristics to find optimal points.

4.2 Protocol Design

The logical implementation of the resource allocation protocol is developed offline in Prolog using the programming environment included in the Event Calculus plugin.

The protocol begins with the participants offering resources to be centrally pooled and allocated by the chair of the session. These offers may or may not be verified for authenticity before allocating the resources according to either a vote between the participants of the system, or a random selection by the chair. There are four degrees of freedom used to form this dynamic protocol:

- *MinimumRequired* - Places a limit on the least amount of resources that an agent must offer each time cycle in order to participate in the allocation.
- *OfferAudit* - Whether we verify the authenticity of the resource offers made by participants before the distribution. This is important if there is an element of agents which misrepresent their contributions.
- *AllocationMethod* - Whether we submit the allocation to a plurality vote, or if the chair simply assigns resources randomly to participants.
- *VotingRights* - This DoF is only valid if a vote occurs in the first place and refers to how long agents must have been members of the system, before they are granted voting rights.

$$dofs = \{votingRights, allocationMethod, minRequired, offerAudit\}$$

$$\begin{aligned} votingRights &= \{trialVote, defaultVote\}, \\ offerAudit &= \{auditOffers, noAudit\}, \\ allocationMethod &= \{plurality, random\}, \\ minRequired &= \{minOffer, noMin\}. \end{aligned}$$

By selecting a value for each DoF, we form a complete specification instance which is referred to as a specification point (SP). Here each DoF can take one of two values, resulting in sixteen possible SPs. If we consider the specification as a state transition system, these points represent all possible states and the adaptations correspond to the transitions between them.

These DoFs must be defined in conjunction with the event calculus implementation of the object level protocol. We present the predicates from the resource allocation example representing how the *allocationMethod* degree of freedom is implemented. The following can be translated directly into Prolog and represents the part of the protocol where the chair is permitted to distribute the pooled resources. $R1a$ refers to a plurality distribution, $R1b$ random allocation. The conditions in bold refer to which of the DoF values is currently active, to ensure that the correct rule is used.

$R1a : holdsAt(permission(Chair, distribute(Chair, Agent)) = true, T) :-$
$holdsAt(roleof(Chair, chair) = true, T),$
$selectPluralityResult(Result),$
$holdsAt(ballot = closed),$
$\boldsymbol{holdsAt(dof(votingProtocol) = plurality,\ T)}.$
$R1b : holdsAt(permission(Chair, distribute(Chair, Agent)) = true, T) :-$
$holdsAt(roleof(Chair, chair) = true, T),$
$selectRandom(Result),$
$holdsAt(ballot = closed),$
$\boldsymbol{holdsAt(dof(votingProtocol) = random,\ T)}.$

The meta level protocol is defined alongside the object protocol in the EC and can be invoked at any time by the agents to initiate a discussion about which DoFs to change in the next session. Note that we have not included any DoFs at this level as we would need another discussion protocol at the meta-meta level to adapt it.

4.3 Metric Space Design

Once the DoFs and their respective values have been set for the protocol a system designer can move from the Event Calculus Plugin to the Metric Space Plugin where the DoF values can be formed into a specification space. Given the set of specification points T determined by the number and value-ranges of the DoFs, a metric space on this set is defined by a distance function d such for any $x, y, z \in T$, d obeys symmetry ($d(x, y) = d(y, x)$), identity of indiscernibles ($d(x, y) = 0 \leftrightarrow x = y$), and the triangle inequality ($d(x, z) \leq d(x, y) + d(y, z)$), from which it follows that for all x and y, $d(x, y) \geq 0$.

To use the **Pre**Sage-$\mathcal{MS}$ plug-in, the programmer must assign the DoF values, and encode the function d. The topology of the space is therefore determined

by the values and d, which are dependent on what the distance between specification points actually represents. If, for example, the distance was a simple similarity measure, we could assign DoF values as follows:

$$\begin{aligned} \mathit{AllocationMethod} &: \{\mathit{plurality}, \mathit{random}\} \rightarrow \{0, 1\} \\ \mathit{VotingRights} &: \{\mathit{defaultVote}, \mathit{trialVote}\} \rightarrow \{0, 1\} \\ \mathit{AuditOffers} &: \{\mathit{offerAudit}, \mathit{noAudit}\} \rightarrow \{0, 1\} \\ \mathit{MinimumRequired} &: \{\mathit{minOffer}, \mathit{noMin}\} \rightarrow \{0, 1\} \end{aligned}$$

and a metric d_1 simply implemented as the hamming distance between points represented as strings in $(0|1)^4$, or the Manhattan distance between cells in a Karnaugh map of the points.

Alternatively, we could define the metric space to represent the *cost* of implementing a (move to a) new specification point. If, in order to change to the new SP, a DoF value must be changed, we define a cost to remove the current value and a further cost to install the new value. Each value must therefore have a cost of installation orthogonal to the cost of removal (resp. installation) of the alternative DoF value. To preserve symmetry, let us assume the cost to install a value to be the same as that to remove it.

Then we could assign values as follows (based on an estimate of the number of lines of code and predicates that have to be changed from the previous section):

$$\begin{aligned} \mathit{AllocationMethod} &: \{\mathit{plurality}, \mathit{random}\} \rightarrow \{(0.5, 0), (0, 0.6)\} \\ \mathit{VotingRights} &: \{\mathit{defaultVote}, \mathit{trialVote}\} \rightarrow \{(0.4, 0), (0, 0.9165)\} \\ \mathit{AuditOffers} &: \{\mathit{offerAudit}, \mathit{noAudit}\} \rightarrow \{(1, 0), (0, 2)\} \\ \mathit{MinimumRequired} &: \{\mathit{minOffer}, \mathit{noMin}\} \rightarrow \{(1, 0), (0, 1)\} \end{aligned}$$

The two specification points in Figure 4 can therefore be encoded as follows:

$$(\overbrace{0,\ 0.6,}^{\mathit{random},}\ \overbrace{1,\ 0,}^{\mathit{offerAudit},}\ \overbrace{0,\ 0.9165,}^{\mathit{trialVote},}\ \overbrace{0,\ 1}^{\mathit{noMin}})$$

$$(\overbrace{0.5,\ 0,}^{\mathit{plurality},}\ \overbrace{1,\ 0,}^{\mathit{offerAudit},}\ \overbrace{0,\ 0.9165,}^{\mathit{trialVote},}\ \overbrace{0,\ 1}^{\mathit{noMin}})$$

Then we need to define a metric d_2 on this space which is the summation of the manhattan distances moved in each of the four dimensions (illustrated in Figure 5). Since we have assigned a cost of 0.5 to implement or remove a plurality allocation method and a cost of 0.6 to implement or remove a random allocation method, then to move from one method to the other the complete removal and installation cost is $0.5 + 0.6 = 1.1$. Similarly to change the voting rights DoF costs $0.4 + 0.9165 = 1.3165$.

Figure 4 shows the formulated metric space given this configuration. The distance between the current specification point (the square node) and the selected circular node in bold is 1.1. This is because the current specification point uses a plurality allocation method and the other vertex is identical except for the random allocation method.

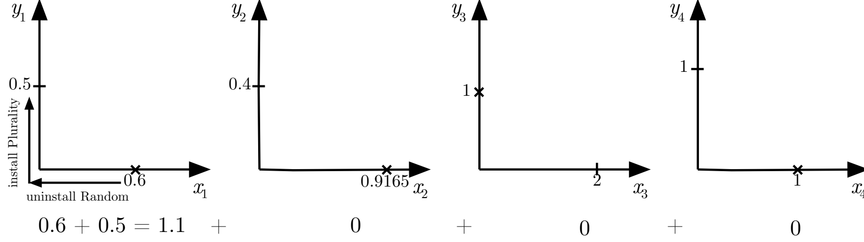

Fig. 5. The distance between the specification points *(trialVote, offerAudit, random, noMin)* and *(trialVote, offerAudit, plurality, noMin)*. Note that we are only changing the allocationMethod degree of freedom from random to plurality

4.4 Experimental Design and Example

Once the protocol specifications are given and the metric space functions defined, a programmer can experiment with effect of the metric on this space for organised adaptation under different agent populations. Given the experimental arrangement described in this paper, we intend to investigate *when a system should adapt*. Agents performing adaptation must be capable of trading off the costs of adaptation versus the expected benefit. This requires that agents can take two measurements: the average utility per timecycle of a specification point under a particular environmental state and an estimate of the amount of time the system will remain in that state. If the cost to adapt consistently outweighs the incentives then the participants must know to refrain from adaptation.

To begin this process we must first limit the environmental measurements to a finite state space. For the resource allocation scenario we have chosen four environmental states, based on the supply and demand of resources and the proportion of agents in the system behaving selfishly:

Env1: Supply > Demand, Few agents behaving selfishly
Env2: Supply < Demand, Few agents behaving selfishly
Env3: Supply > Demand, Many agents behaving selfishly
Env4: Supply < Demand, Many agents behaving selfishly

We can then run non-adaptive simulations for each environmental state under each of the twelve specification points to get the expected utilities:

$$\begin{array}{c} \\ env1 \\ env2 \\ env3 \\ env4 \end{array} \begin{array}{c} \begin{array}{cccc} sp1 & sp2 & \dots & sp12 \end{array} \\ \begin{pmatrix} \mu_{1,1} & \mu_{1,2} & \cdots & \mu_{1,16} \\ \mu_{2,1} & \mu_{2,2} & \cdots & \mu_{2,16} \\ \mu_{3,1} & \mu_{3,2} & \cdots & \mu_{3,16} \\ \mu_{4,1} & \mu_{4,2} & \cdots & \mu_{4,16} \end{pmatrix} \end{array}$$

If we assume that the utility is only dependent on the current state we can calculate the total expected benefit by estimating the time the system will remain in an environmental state. For the preliminary experiments we can take a simple geometric distribution of rate λ to predict the number of timecycles between state changes. The expected utility of $sp1$ in under $env1$ then becomes:

$$E_{env1}(sp1) = \frac{\mu_{1,1}}{\lambda}$$

Given these values we can form a decision function based by amending each of the utility values with their associated cost. This cost is calculated from the second metric space formulated in section 4.3. Based on this decision function, we intend to look at different movement policies through specification spaces in order to find which ones contribute to a more desirable distribution of resources.

5 Discussion, Related and Further Research

PreSage-$\mathcal{MS}$ is proving useful for experimenting with dynamic specifications in terms of the three parameters: agent internals, protocol and DoF specification, and metric space design. However, it has also thrown into relief a number of inter-linked issues concerning organised adaption, namely:

- the role of institutional agents and the migration from design-time tools to run-time services;
- scalability to 'large' agent teams and the transition from organised adaptation to organizational adaptation; and
- the relationship between formal models of norm change and compliance pervasion amongst the affected population of agents.

We briefly consider these issues here.

Open systems, following Hewitt [9], assume independently developed subsystems without global objects or objectives, but with a commonly understood communication language. However, it is possible to relax the assumption that there are no global objects, and consider the status of the EC plug-in and the Metric Space plug-in. These plug-ins are providing computationally-intensive services; they are also computing a storing a global state (the set of norms and normative positions, and the current/desired specification points). In these cases, we may wrap such services in institutional agents of the kind envisaged by Lopez et al [4]. As such, this move represents a step in the migration from design-time tools for human user to run-time services for software agents.

The presence of institutional agents also suggests a possible repository for the functionality required to affect the transition from organised adaptation to organizational adaptation. Recall that a design objective was adaptation in 'large' agent teams: the question then is whether it is possible (necessary, desirable) to have a single flat hierarchy, or whether some kind of structure is required, e.g. [8]. In organizational adaptation, it is needed to extend a dynamic specification to the creation, modification and deletion of roles, and the creation, remit, modification and deletion of sub-structures (in the way that human institutions are

often structured into departments, committees, etc.) This opens up an inquiry into the scale and effective size of 'sophisticated' agent teams, and whether some form of Dunbar's number can be derived for agents (i.e. a computational rather than a cognitive limit on the size of a stable group).

Many other issues are raised by stable groups in organizational adaptation, which includes the impact of an underlying social network on the 'observable' institutional structure (cf. [3]). This includes: how an arbitrary collection of agents can self-organise into an organisational structure; how an arbitrary collection of agents can self-organise its social network (which is structurally distinct from the organizational structure); and, what is the interplay between the explicit formal organization and the implicit social network.

The consideration of social networks as a parameter influencing adaption raises additional concerns with respect to the relationship between formal models of norm change and compliance pervasion amongst the affected population of agents. There is a growing body of work on formal models of explicit norm change in legal systems [7]. A particular question for future research is how to use **Pre**Sage-$\mathcal{MS}$ to experiment with modifiable legal systems and synthetic micro-populations of agents and the interaction between the two: i.e. how does ('top down') norm change impact population behaviour, and how does population behaviour influence ('bottom up') changes to norms.

6 Summary and Conclusions

The system represents a novel design framework for open systems performing organisational adaptation. The rules and protocols are entirely specified in the Event Calculus providing foreign agents with a transparent description of how the system functions and as such can decide whether their interests will be served. Metric Spaces are used in conjunction with a variety of plugins to manage which aspects of the system may be adapted in order to prevent an adaptation which could irreparably harm the system. It is in this way that we ensure the level of control required by a system designer while at the same time maintaining a publicly accessible specification.

PreSage-$\mathcal{MS}$ is an extended agent programming environment which offers a flexible and open solution to implementing organised adaptation in multi agent systems using dynamic specifications. We intend to explore the question of *when to adapt* in the context of a self adapting resource allocation scenario in which the agents stockpile and allocate a shared resource. Periodically the system will need to adapt to deal with invasions by a selfish population who try to divert the flow of resources to themselves. We intend to explore movement policies in specification spaces, which trade off the cost of making an adaptation versus the benefit of moving to a different specification point.

References

1. Michael Apostolou and Alexander Artikis. Evaluating dynamic protocols for open agent systems. *AAMAS'09: Proceedings of The 8th International Conference on*

Autonomous Agents and Multiagent Systems, pages 1419–1420, May 2009.
2. Alexander Artikis. Dynamic protocols for open agent systems. *AAMAS'09: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, pages 97–104, 2009.
3. Michael Ashworth and Kathleen Carley. Who you know vs. what you know: The impact of social position and knowledge on team performance. *Journal of Mathematical Sociology: Journal of Mathematical Sociology*, 30:43–75, Jan 2006.
4. Eva Bou, Maite Lopez-Sanchez, and Juan Rodriguez-Aguilar. Adaptation of autonomic electronic institutions through norms and institutional agents. *ESAW'06: Proceedings of the seventh annual international workshop on engineering societies in the agents world*, LNCS 4457:300–319, 2007.
5. Hugo Carr and Jeremy Pitt. Adaptation of voting rules in agent societies. *OAMAS@AAMAS'08: Proceedings from the AAMAS Workshop on Organised Adaptation in Multi-Agent Systems*, pages 36–53, 2008.
6. Hugo Carr, Jeremy Pitt, and Alexander Artikis. Peer pressure as a driver of adaptation in agent societies. *ESAW'08: Proceedings of the ninth annual international workshop Engineering Societies in the Agents World*, LNCS 5485:191–207, 2008.
7. Guido Governatori, Antonino Rotolo, Régis Riveret, Monica Palmirani, and Giovanni Sartor. Variants of temporal defeasible logics for modelling norm modifications. *ICAIL'07: Proceedings of the 11th international conference on Artificial intelligence and law*, pages 155–159, Jun 2007.
8. Zahia Guessoum, Mikal Ziane, and Nora Faci. Monitoring and organizational-level adaptation of multi-agent systems. *AAMAS'04: Proceedings of the third international joint conference on autonomous agents and multiagent systems*, 2:514 – 521, Jul 2004.
9. Carl Hewitt. Offices are open systems. *ACM TOIS: ACM Transactions on Information Systems*, 4(3):271–287, Jul 1986.
10. Owen Holland and Chris Melhuish. Stigmergy, self-organization, and sorting in collective robotics. *Artificial Life*, 5(2):173–202, Jan 1999.
11. Bevan Jarvis, Dennis Jarvis, and Lakhmi Jain. Teams in multi-agent systems. *International Federation for Information Processing (Springer)*, 228:1–10, Jan 2007.
12. James Kennedy, Russell C Eberhart, and Yuhui Shi. Swarm intelligence. *Springer*, Jan 2001.
13. Jeffrey Kephart. Research challenges of autonomic computing. *ICSE'05: Proceedings of the 27th international conference on Software engineering*, pages 15–22, May 2005.
14. Robert Kowalski and Marek Sergot. A logic-based calculus of events. *New generation computing*, 4(1):67–95, 1986.
15. Brendan Neville and Jeremy Pitt. Presage: A programming environment for the simulation of agent societies. volume LNCS 5442, pages 88–103, 2008.
16. Mícheál Ó Searcóid. Metric spaces. *Springer*, Jan 2006.

Action and Perception in Multi-Agent Programming Languages: From Exogenous to Endogenous Environments

Alessandro Ricci, Andrea Santi, and Michele Piunti

DEIS, Alma Mater Studiorum – Università di Bologna
via Venezia 52, 47023 Cesena, Italy
{a.ricci,a.santi,michele.piunti}@unibo.it

Abstract. The action and perception model adopted by current multi-agent programming languages has been conceived to work with *exogenous* environments, i.e. physical or even computational environments completely external to the multi-agent system (MAS) and then out of MAS design and programming. In this paper we discuss the limits of adopting such models when *endogenous* environments are considered, i.e. fully computational (software) environments that are used by MAS developers as first-class abstraction in MAS engineering to encapsulate functionalities useful for, e.g., agent coordination, agent computational activities and agent access to the external environment. In the paper we describe an action and perception model for agent programming languages specifically conceived to be effective for endogenous environments and we discuss its evaluation using CArtAgO environment technology. On the agent side, we take Jason, 2APL and GOAL as reference agent programming languages.

Categories and subject descriptors: I.2.5 [**Artificial Intelligence**]: Programming Languages and Software; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Intelligent Agents*

General terms: Agent programming languages; Methodologies and Languages

Keywords: Environment programming, Action/perception models

1 Introduction

The context of this paper is the action and perception model adopted in agent programming languages (APL). Here we refer in particular to programming languages for (multi-) intelligent agent systems, based on practical reasoning: in this paper in particular we will consider Jason [3], 2APL [4], and GOAL [7] as concrete cases. The action and perception model in such abstract architecture (and concrete languages) have been originally devised to be effective for agents situated in *physical* environments, or computational but – in any case – *external*

to the multi-agent systems (MAS), not part of MAS design and engineering. This is also the main perspective adopted in AI (see Russel and Norvig [13]). We refer to environments of this kind as *exogenous*.

Besides this exogenous characterization, a further notion of environment has been introduced, more oriented to MAS engineering: environment as *first-class abstraction* [14], that is a computational layer which is meant to be designed by MAS engineers to encapsulate functionalities that agents can exploit and adapt at runtime, for their purposes. Functionalities range from – simply – providing an interface to exogenous environments, to making it available computational resources and services that are useful for agents individual activities, up to providing functionalities that can make agents communication, coordination and organization more effective [10, 9]. In literature such a kind of environments take different names (e.g. application environment, working environment): here we refer to environments of this kind as *endogenous*, since they are part of MAS. In this perspective, agent programming languages on the one side and technologies for programming endogenous environments on the other side, can be suitably integrated to develop intelligent software systems [11], keeping a strong separation of concerns by using the former to program pro-active, goal-oriented parts of the system and the latter to program the passive resources and facilities shared and co-used by the goal/task oriented parts.

Clearly the action/perception model adopted plays a key role in such integration, being the *interface* between the agent and environment dimensions. Actually, almost every APL includes some kind of API to define the interface to the environment. In this context, the EIS (Environment Interface Standard) initiative [1] aims at defining a standard interface to allow agents developed using different programming languages to share the same environment, independently of the specific model and technology adopted for it. Furthermore, languages like 2APL, Jason and GOAL provides a basic Object-Oriented model to define the environment itself (besides the interface), which can be programmed using the Java language. In this paper e argue that the action and perception model currently adopted by APL, which is suitable for exogenous environments, is not fully satisfactory when endogenous environments are of concerns. The weaknesses – that will be discussed in detail in Section 2 – concern both the approach used in agent architectures to keep track of the actual state of the environment and the semantics adopted for actions and action execution, which are not expressive enough to effectively exploit endogenous environment mechanisms, such as synchronization. Such problems can have a significant impact on both agents and environment programming, as well as on systems scalability. Accordingly, in order to solve the problems encountered with current approaches here we propose a model for perceptions and actions specifically conceived for agents working in endogenous environments. To evaluate the approach, we implemented it in a new version of CArtAgO, which is a technology for programming and executing endogenous environments in MAS based on the *artifact* abstraction [12, 8].

The remainder of the paper is organized as follows. In Section 2 we analyze the different kind of action/perception models in current APL, taking as

references Jason, 2APL and GOAL, and we describe what are the issues when adopting such models in endogenous environments. In Section 3 we describe the main concepts underlying a revised action/perception model and in Section 4 we evaluate its implementation by discussing some examples developed using the new version CArtAgO and Jason as reference APL. Finally, conclusions and future works are provided in Section 5.

2 Action and Perception in Agent Programming Languages

In this section we analyze and discuss the action and perception model and related architectures adopted in current agent programming languages, taking Jason, 2APL and GOAL as reference case studies.

By referring to existing formalizations, all these languages follow more or less the abstract reference architecture for intelligent agents and the practical reasoning agent cycle reported by Wooldrige in [15]. Essentially such a control loop can be sumarized as a *sense-plan-act* cycle where the agent repeatedly *(i)* observes the environment and update its beliefs, *(ii)* uses practical reasoning to deliberate what intention achieve and how, and *(iii)* executes a proper plan for fulfilling the selected intention. The environment (software or hardware) here is considered fully exogenous. It's worth noting that, moving from formal models to concrete architectures and implementations, current APL adopt richer approaches and semantics, which are explicitly oriented toward the integration with forms of endogenous environments, typically developed using a mainstream language such as Java. A comprehensive survey of the environment interface models adopted by mainstream APL and the API for interact with them are discussed in the EIS initiative report [1]: here we focus on the semantics underlying the action and perception model.

2.1 The action model

In the abstract architecture, the action chosen by agent's action selection function is dispatched to effectors which will eventually execute it (act stage, or *execute* command in the practical reasoning cycle) and the control cycle can start again (sense stage). Actions are considered as options in agents repertoire which can be translated by moves enabled by the environment. The success or failure of the action executed by effectors is meant to be determined by an agent by analyzing the percepts that will eventually be observed from the environment. From the execution model point of view, action execution is modelled then as an atomic *event*, which corresponds to dispatching the action to effectors. This semantics is the basic one adopted by almost all APLs formal models. Not surprisingly, concrete implementations of APLs adopt more complex solutions than the one just presented.

In AgentSpeak(L) and Jason operational semantics [2], action execution is modelled by a transition inserting the action selected by previous stages of the

agent cycle into a particular set of actions, i.e. a set of actions to be performed in the environment. The formal model does not provide any further information: the selected action is scheduled to be executed – sooner or later – by other components (i.e. effectors) of the agent architecture [2]. Actually, in the concrete architecture and implementation of the Jason interpreter[1], the action execution model is more articulated and expressive than the one described in the operational semantics. Action execution is done by calling a special method of the Java class representing the environment—the action to be performed is a parameter of this method. As a key aspect, the method is executed *asynchronously* with respect to the agent cycle: current agent intention – i.e. the plan in execution – is *suspended* until the action execution is terminated, so the agent can carry on other plans and react to percepts. So, in practice, the action execution model is not atomic. The environment method can return – as action feedback – a boolean value, indicating if the requested action has been executed at all. A false value means it was not, so the plan fails. A true value means that the action as been executed (accepted), however it does not mean that the expected changes will necessarily take place ([2], p. 50).

Also GOAL adopts the basic action semantics found in the abstract practical reasoning agent cycle, so executing the actions atomically and establishing their outcomes only by sensing the environment. Analogously to Jason, action execution is done by calling a special execute-action method of the Java class representing the environment, with the action to be performed as a parameter of this method. As reported in [1], in GOAL invoking the execute-action method might have tree outcomes: either the return value true indicating success, false indicating that the action has not been recognized, or an exception indicating that the action has failed. In this case the meaning of *success* is more subtle: it triggers the application of the action post-conditions that can be specified in the agent program, to update the belief base. In the examples described in [7] – the blocks-world in particular – such updates seem to refer not simply to an action (e.g. move) that has been recognized by the environment, but to an action whose execution has been fully executed with success and the environment has been changed accordingly.

So, this action semantics appears a natural choice when exogenous environments are considered – even if also in this case some criticisms have been raised in literature [5]. Conversely, we argue that it is not the most effective semantics when considering endogenous environments. The reason is that endogenous environments are meant to be specifically designed and programmed by MAS developers to support agent activities; so, it is natural to devise a stronger semantics for action execution where – for instance – action success, and so the success of the execute-action method if we consider the environment interface model adopted by Jason[2] and GOAL – means not only that the action has been *accepted* or *recognized* by the environment, but that it has completed and its

[1] `http://jason.sourceforge.net`

[2] Actually Jason makes it possible to implement a different semantics by customizing some part of the agent architecture.

```
// agent code (Tom)

Goals:
  do_job

PG-rules:
 do_job <- true |
 {  @TestEnv(compute(100000), Res );
    @TestEnv(log(Res));
    dropgoal(do_job)
  }

PC-rules:
 event( my_ev(Num), simple2APLEnv)
                              <- true |
 {
   @TestEnv(log(Num))
 }
```

```
public class TestEnv extends Environment {

 public APLNum compute(String agent, APLNum howLong)
 {
   long sum = 0;
   int burn= howLong.toInt();
   for (int i = 0; i < burn; i++){
     for (int j = 0; j < 100000; j++){
        sum+=i*j;
    }
   }
   return ( new APLNum( sum ) );
 }

 public void update(String user, APLNum n)
 {
   int nEvents = n.toInt();
   for(int i=0; i < nEvents; i++){
     APLNum evNum = new APLNum(i);
     APLFunction event = new APLFunction( "my_ev", evNum );
     throwEvent(event);
   }
 }

 public void log(String user, APLString s){
   System.out.println(s);
 }
}
```

Fig. 1. *(left)* 2APL example for an agent blocked on a long term activity. *(right)* A simple 2APL environment in Java.

effects and changes took place, relieving the agent from the burden of checking this by analyzing the percepts.

2APL apparently goes in this direction, by adopting a strong semantics for action success and failure, not only at the implementation level but also in the formal model, explicitly assuming to work with computational environments, represented by Java objects. In the formal model, an external action is modeled as an atomic transition involving changes in the configuration of both the agent and the environment [4]. Action execution is done by calling methods of the Java class(es) representing the environment(s), which directly implement the actions behaviour. Following [1], executing an action-method in 2APL can have two outcomes: either a return-value (an object) indicating success is returned, that might be non-trivial (e.g. list of percepts in the case of an sense-action) or terminate with an exception indicating action-failure. In this case success means that the action completed with success—and so the effects of the action execution took place, so a stronger semantics with respect to Jason and GOAL. However, analogously to Jason and GOAL formal model, action execution is modelled and finally implemented as an atomic transition (so an event) coupling the agent and the environment. This means that by executing an action, the agent cycle is blocked until the completion of the action with success or failure occurs, and this can have some drawbacks for agent reactivity. We clarify the point with an example shown in Fig. 1, a 2APL program composed by two agents, Tom and Alice, interacting in the same environment, represented by the `TestEnv` Java class (only Tom's source code is shown). The actions provided by the environment – i.e. the methods implemented by the class – are `compute`, which is meant to execute a long-term computation returning finally a result – and `update`, which is meant to update the state of the environment generating percepts—that are

events in the case of 2APL. To achieve its goal, the agent Tom must perform the `compute` action; however Tom is also interested to perceive events generated by the environment to react accordingly—in this case simply logging in output the percepts. Alice simply acts on the environment performing an `update`, so generating percepts that are relevant also for Tom. However Tom *is not able to react to percepts generated by the environment until the* `compute` *action has completed*. So if one want to save agent reactivity, it cannot perform long-term actions, or rather: every long term action must be implemented by environments in terms of multiple sub-actions.

2.2 The perception model

In the abstract intelligent agent architecture [15], agent perception is modeled by a *see* function: $E \rightarrow Per$. This function encapsulates agent's ability to obtain information from the environment E in which it is situated. The output produced by this function is a percept Per, which typically contains information about the *actual state* of the environment. Percepts are then elaborated by the agent through appropriate belief-update/revision functions, to keep its mental state consistent with the actual state of the environment. This model is adopted both in Jason and GOAL.

A GOAL agent implements a simple SPA cycle during which (*i*) it receives percept from the environment (containing the whole state of interest) through its perceptual interface and (*ii*) it updates its mental state through the percepts rules included by the agent programmer to specify how the agent belief base should be updated when certain percepts are received (see [7] for more details). In Jason, at each cycle an agent perceives the actual state of the environment and updates its mental state, in particular automatically removing / updating / adding beliefs related to percepts. Actually this is the default behaviour of the belief-update and belief-revision function: the highly customizable architecture of the Jason interpreter allows for changing this semantics, customizing both the perceive stage (implementing the *see* function) and the belief update/revision stage.

So both in the case of GOAL and Jason, percepts represent *actual* information regarding the environment—typically a snapshot of the environment state which is observable to the agent. This is the natural choice when considering exogenous environments. However, it suffers of some problems that are particularly important when applying the approach to endogenous environments. A first problem concerns the possibility for an agent of *losing* (not perceiving) environment states that could be relevant for agent reasoning and course of action. This can occur because of environment dynamics, related to internal processes and also actions performed by other agents, changing asynchronously the environment multiple times between two subsequent perceive stages. We clarify the problem with a simple example written in Jason, shown in Fig. 2. The example includes a simple endogenous environment that provides a generic *shared resource* bounded by capacity limit, i.e. the resource can be used concurrently only by a limited number of agents. Then, a set of `worker` agents – with a cardinality greater than the

```
// worker source code

times(100).

!run_worker.
+!run_worker : times(X) & X \== 0
  <- acquire_resource;
     !use_resource;
     release_resource;
     -+times(X-1);
     !!run_worker.

-!run_worker : true
  <- .wait(10);
     !!run_worker.
```

```
// Observer agent source code

last_occurrence(0).

@my_plan [atomic]
+max_use(Occurrence) : last_occurrence(N)
  <- .print("New max usage perceived:",
            Occurrence);
     !check_state_lost(Occurrence, N);
     -+last_occurrence(Occurrence).

+!check_state_lost(Current,Local): true
  <- X = Local + 1;
     if (Current > X){
       .print("State(s) lost! from ",
              Local," to ",Current)
     }.
```

```
// Jason Environment

public class ResEnv extends jason.environment.Environment {
  int occurenceNum = 0;
  int resourceUse = 0;
  int maxResourceUse = 3;
  ReentrantLock lock;

  public void init(String[] args) {
    ReentrantLock lock = new ReentrantLock();
  }

  public boolean synchronized executeAction(String ag,
                                            Structure act) {
    if (act.getFunctor().equals("acquire_resource")){
      if (resourceUse == maxResourceUse){
        return false;
      } else{
        resourceUse++;
        if (resourceUse == maxResourceUse){
          clearPercepts();
          occurenceNum++;
          String perc = "max_use("+occurenceNum+")";
          addPercept(Literal.parseLiteral(perc));
        }
      }
    } else if (act.getFunctor().equals("release_resource")){
      if (resourceUse == 0) {
        return false;
      } {
        else resourceUse--;
      }
    }
    informAgsEnvironmentChanged();
    return true;
  }
}
```

```
MAS Console - test_losing_states
[observer] New max usage perceived: 4
[observer] New max usage perceived: 5
[observer] New max usage perceived: 7
[observer] State(s) lost! from 5 to 7
[observer] New max usage perceived: 8
[observer] New max usage perceived: 27
[observer] State(s) lost! from 8 to 27
[observer] New max usage perceived: 28
Clean  Stop  Continue  Debug  Sources  New agent  Kill agent
```

Fig. 2. *(left)* Jason source code of the worker and observer agents. *(right)* Java implementation of the resource environment where the workers and observer are situated. . *(bottom)* A screenshot of the Jason console during a run of a MAS composed by three workers and an observer.

resource capacity – cyclically try to use the resource, first attempting to acquire tit by means of an `acquire_resource` action, then using it and finally releasing it by means of a `release_resource` action, making it available for further usages. The single kind of percept generated by environment – `max_use(N)` – represent the actual number of times that the resource reached its max capacity—so it starts from zero and is incremented each time the limit is achieved. The scenario is completed by an `observer` agent, whose goal is to observe the shared resource and to log in output a message each time `max_use` changes. The agent monitors the number of times the resource reached its max capacity and, also, does a check for detecting lost states, by executing the `check_state_lost` plan each time it perceives a new value for the max use – represented by the belief-update event related to `max_use(N)`. The `check_state_lost` plan checks if the difference

between the current observed `max_use` occurrence and the previous one (stored by the `observer` into the `last_occurence` belief) is greater than one unit. If that is the case, it means that some `max_use` percepts have been lost and a message is printed in standard output. By running the example it is possible to verify that the observer loses states (Fig. 2 shows a screenshot) and that the frequency of the losses increases with the number of worker agents concurrently working in the same environment.

A second problem is that retrieving perceptual information at each cycle regarding the current observable state of the environment can be a task that requires high computational complexity, in particular when considering non-naive environments, being either centralized or, worst, distributed. Then, given the list of percepts representing the current state, the belief update must be updated accordingly: in the case of a direct mapping between beliefs and environment state as in the case of Jason adopting the default belief-update and belief-revision semantics, this typically involves iterating both the percept list and the whole belief-base to remove belief that does not hold anymore, to add new ones and update existing ones.

Differently from Jason, 2APL models percepts as events. This in principle makes it possible to avoid the previous problem—as will be shown in next section. However in 2APL – analogously to GOAL – in order to keep track of the observable state of the environment in terms of beliefs, the programmer is forced to explicitly define the rules that specify how to change the belief-base when a percept is detected. This is a indeed very important capability when dealing with exogenous environments; when adopting endogenous environments – in particular complex ones – this can become burdensome. Actually – as will be detailed in next section – being specifically designed by MAS engineers, endogenous environments allow for stronger assumption on the relationships between percepts generated by the environment and related beliefs. Such assumption can be used then to define a mapping percepts-beliefs to be applied by default by the architecture, so, on the one side, avoiding the burden to the agent programmers of necessarily specifying the percept rules, and on the other side automatically keeping consistency between the actual state of the environment and the belief base.

3 From Exogenous to Endogenous Environments

3.1 Action side

As mentioned in the previous sections, endogenous environments allow for a stronger semantics for action success/failure, which finally simplifies agent programming and make multi-agent programs more efficient. In an endogenous environment the success (or failure) of an action on the agent side can be directly related to the successful (or failed) completion of an operation or process executed on the environment side – which has been designed by the MAS engineers – as a consequence of the agent action request. So, differently from exogenous environments, where action success or failure can be established by an agent

only by interpreting the percepts generated by the environment after the action execution, in endogenous environments action success/failure can be represented by an explicit *action completion event* generated by the environment, thereby an explicit information related to operation execution completion (with success or failure). Accordingly, from the APL point of view the execution of an action does not mean that the action has been simply accepted or recognized by the environment, but that the related environment operation has been executed up to completion. This means seeing the set of actions as a sort of *contract* provided by the environment. The contract includes both the *effects* that can be assumed with action completion, and the *action feedbacks*, including further information (results) related to action success or failure. By assuming this semantics, agent programs become – generally speaking – more compact and efficient, since there is no need for agents to check asynchronously percepts value to determine action effects.

From the action execution model point of view, this approach promotes an *action-as-a-process* semantics, where actions are not modeled as single atomic events but as processes that can be long-term, whose completion is notified by action completion events. When adopted in APL, this semantics have two main benefits: First, it makes it possible to effectively program agents that execute (long-term) actions without hampering their reactivity (see the example using 2APL in Section 2): the action-as-process semantics makes it possible then for an agent to start the execution of action and then go on perceiving, reacting to percepts that are generated by the action itself or other actions, possibly carrying on other pro-activities by choosing other actions to execute. Second, the action-as-process semantics makes it possible to implement efficient *coordination mechanisms* simply based on action synchronization, designing environments which provide operations for that purpose. This because the action completion event of an action performed by a certain agent can be generated as a consequence of the execution of the action(s) of other agents in the same environment. This is actually not possible with an action-as-event semantics. The coordination semantics in this case is encapsulated in the the environment providing the operations. As a well-known example, we mention here tuple space coordination model and Linda coordination language [6]. Blocking actions like in or read cannot be implemented by adopting an action-as-event semantics: conversely their implementation is quite straightforward by considering an action-as-process semantics. A concrete example about this will be given in next section, using CArtAgO.

3.2 Perception side

On the perception side, in endogenous environments modeling percepts as *events* – carrying on information about *changes* occurred in the environment – is more effective and efficient than modeling percepts as facts about the *actual state* of the environment itself, as in the case of exogenous environments. At a first glance this makes it possible to solve a main problem that has been described in Section 2, i.e. the possibility for agents not to perceive environment states – so loosing relevant environment configurations – due to the different update

frequencies carried on by agents perceptive activities and environment internal processes. Referring to the abstract model mentioned in Section 2, the set of percepts returned by the *see* function represent a list of changes occurred in the environment during the last agent execution cycle and which are relevant for the observing agent. Accordingly, the *next* function can update the current internal state of the agent with respect to the whole set of changes occurred inside the environment, thus eventually reconstructing all the intermediate states that the environment assumed between a couple of *see* activities .

Then, to support the automated reconstruction of such states it is useful to identify the basic set of possible kinds of event that can occur inside an endogenous environment: *(i)* an observable part of the environment has changed; *(ii)* an observable part of the environment has been added or removed; *(iii)* a signal has been generated to acknowledge agents with some information. In the latter case, signals are meant as information explicitly generated by the environment – as designed by the environment programmer – to carry on some data which can be purposefully processed by agents situated in the environment and focusing that part of the environment which is the source of the signals.

By explicitly defining a model to represent environment observable parts – for instance *observable properties*, in the case of artifact-based environments used in next section – it is possible then at the agent architecture level to automatically reconstruct a consistent snapshot of the current observable state of the environment by processing a list of events updating the previous snapshot. In the APL considered here this means introducing in the basic architecture a support for *(i)* representing the observable part of the endogenous environments as beliefs, and *(ii)* automatically updating such beliefs as soon as such events are processed. In this perspective, there is no more the need for an agent programmer to explicitly define belief updates function (such as in 2APL) or percept rules and post-condition in action specification: the belief base is automatically updated reflecting the perceived/reconstructed state of the observed environment.

3.3 The importance of defining localities

Actually, due to concurrency and distribution, the correct and efficient reconstruction of the observable state of the environment from the individual agent architecture perspective is an issue, both from the theoretical and practical point of view. First, by working with multi-agent systems, we must assume that multiple agents can concurrently work in the same environment and then events generated concern concurrent processes; Second, environments can be distributed, which means that it is not feasible to consider a priori the availability of a unique notion of time – either physical or logical – and then a total order among events.

In order to cope with these two aspects, first it is useful to conceive a distributed endogenous environment as a set of non-distributed sub-environments, eventually connected in some way, and assume that each sub-environment defines a spatial-temporal locality. For each sub-environment it is feasible then to assume that *(i)* a local logical notion of time can be defined, and *(ii)* observable events occurring the in the sub-environment can be totally ordered using logical

timestamps, even if they are generated by concurrent processes. Given this assumption, agents perceive chains of events, which are totally ordered if the source is a single sub-environment, partially ordered if more sub-environments are involved. Then, some *modularization* strategy should be considered for structuring individual sub-environments, so as to *(i)* allow multiple agents to work concurrently to different parts of the overall structure, promoting as far as possible decentralization and parallelism; *(ii)* make it possible to easily change structure at runtime, eventually changing/extending dynamically the set of actions available, so to better support openness, adaptation, etc.

4 Evaluation using CArtAgO

The idea presented in previous section has been implemented in the new version CArtAgO [12], a platform for developing endogenous environments in multi-agent systems. Before discussing in detail some examples evaluating the new action/perception model, few words about CArtAgO are provided – a complete description is outside the scope of this paper, the interested reader can find it here [12]. CArtAgO makes it possible to design and program endogenous environments as set of *workspaces* – playing the roles of sub-environments – where agents can share and use *artifacts*, which is the basic abstraction used to modularize the environment. Artifacts are programmed by MAS developers – a Java API is provided to this end – and instantiated, used, adapted by agents at runtime as first-class *resources* and *tools* composing their endogenous environment, to be exploited to accomplish their tasks. To be used, an artifact provides a usage interface containing a set of *operations* that agents can execute to get some functionality. To be perceived, an artifact can have one or multiple *observable properties*, as data items that can be perceived by agents as environment state variables, whose value can change dynamically because of operation execution. Operations are computational processes occurring inside the artifact, possibly changing the observable properties and generating *observable events*, as environment signals that can be relevant for agents using/observing the artifact.

By integrating CArtAgO with existing APL, agents written in different agent programming languages can cooperatively work inside the same workspaces, sharing and co-using the same artifacts [11]. The new action and perception model described in this paper essentially improves the way in which agents can exploit artifact-based environments. In the remainder of the section we will consider as APL Jason, whose highly customizable architecture made it possible to adapt quite straightforwardly the agent architecture to implement the new action and perception semantics.

4.1 The action model at work

Following the new model, artifacts operations are now conceived directly as external actions that the environment makes it available. So, by performing an action $act(P)$ where P are action parameters, a corresponding operation $op(P)$

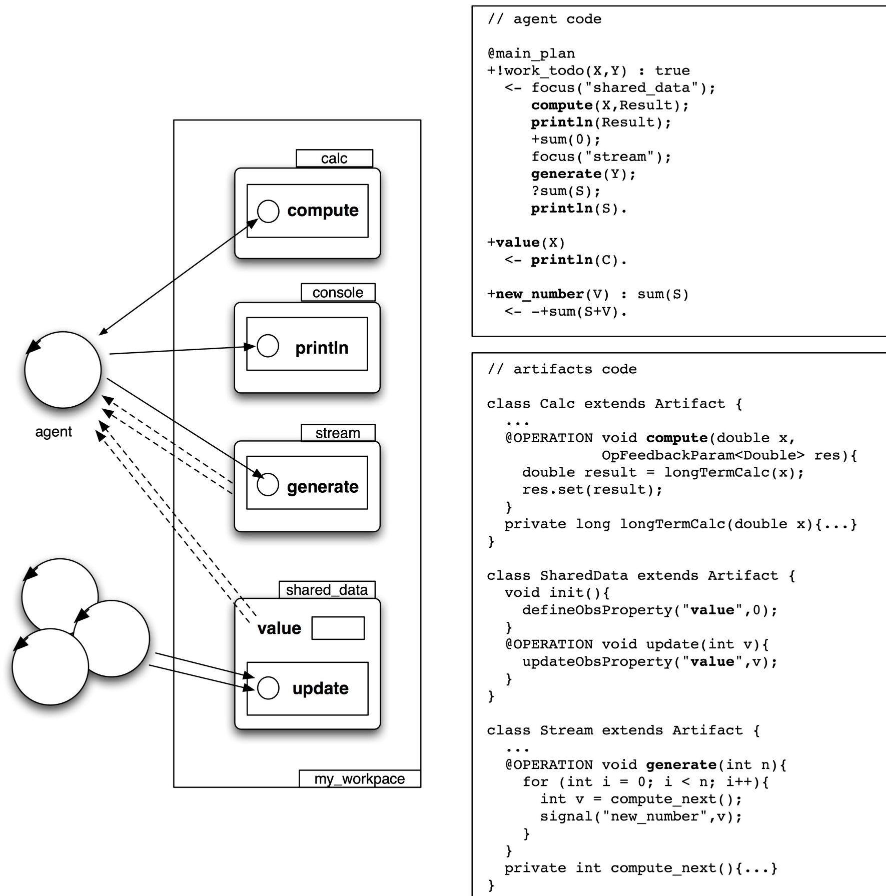

Fig. 3. A Jason agent executing some actions to exploit the functionalities of the artifacts of a workspace, reacting to percepts related to the environment observable state and events.

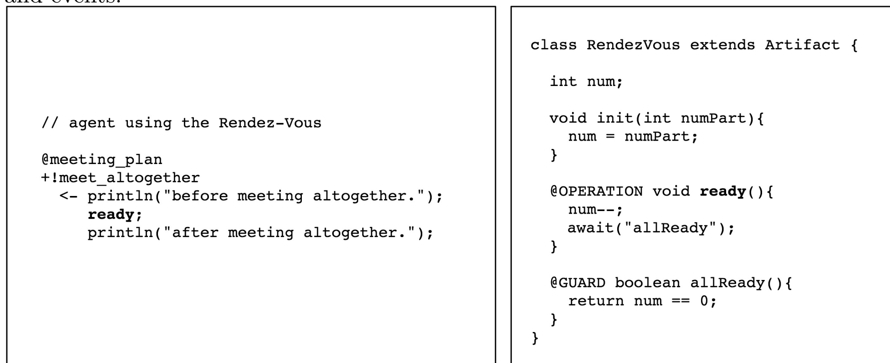

Fig. 4. *(left)* Snipped of a plan of Jason agent to achieve a meeting point with other agents, exploiting a `RendezVous` artifact; *(right)* Source code of the `RendezVous` artifact.

provided by some artifact currently available in the workspace is executed—where *act* and *op* matches. Then, the action succeeds or fails when (if) the corresponding operation has completed with success or failure. Action feedbacks eventually resulting from action execution are made available to the agent performing the action (operation) as output parameters of the action itself. By executing an action, the agent plan (activity) including such action is suspended until the corresponding operation has completed (i.e. the action completed). In the meanwhile, the agent control cycle can go on, making it possible for the agent to get percepts and select and perform other actions. So by adopting this semantics the use of artifact-based environments by agents becomes more agile and agent programs more concise.

As a simple example, Fig. 4 shows a Jason agent working in a workspace containing some artifacts, in particular: an instance of `Calc` artifact, an instance of `SharedData` artifact, one of `Stream` and one `Console`. `Calc` provides an operation `compute` which does a long term computation returning finally a result as feedback. `SharedData` has a `value` observable property and provides an operation, `update`, to update such value. `Stream` has an operation `generate` which results in the generation of a stream of observable events (signals in this case) which can be perceived by agents focussing the artifact. Finally `Console` (whose source code is not reported) provides an operation `println` to print messages on standard output. At a first glance, the agent sees the workspace as an environment providing four kind of external actions: `compute`, `update`, `generate` and `println`, and an observable property `value(X)`, besides the specific artifacts where the operations and observable properties are stored. Accordingly, in `main_plan` plan, which is triggered by a new `work_todo` goal, the agent interacts with the environment directly performing `compute` and then `println` to show the computed result on standard output. Actually, to avoid ambiguities when performing actions in the case of multiple instances of artifacts providing operations with the same names, it is possible to specify the artifact target of the action by means of proper annotations, such as `artifact_name`: `compute(X,Result) [ artifact_name("calc") ]`. By triggering the execution of `compute` – which is carried on asynchronously in the environment – `my_plan` is suspended until the action has completed, reporting the result as feedback (second parameter). Even if this plan is suspended, the agent is free to carry on other plans and react to percepts. In the example the agent – by executing a `focus` at the beginning of the plan[3] – is observing the `shared_data` artifact, whose observable properties (`value`) are mapped then onto the agent belief base. So, as soon as the value of the property changes (because some other agents perform an `update`), the belief is automatically updated and a new belief update event is generated, triggering a plan that simply prints (by exploiting the console) the value on standard output. Finally, after using the `calc` artifact, the agent uses the `stream` artifact by doing a `generate`. Then, by observing the `stream`, the agent processes the events generated by the artifact –

[3] `focus` is a basic primitive of CArtAgO which makes it possible for an agent to select the parts (artifacts) of the workspace to be aware of, perceiving their observable properties and events

by updating a belief related to the sum of the values generated by the stream – before the `generate` action completes. After the action completion, the agent then prints on standard output the final sum.

It's worth noting that by mapping external actions onto artifact operations we have a further important outcome for what concerns openness and dynamism: the set of external actions available to an agent is dynamic, it depends on the current shape of the environment – the actual set of artifacts available in workspaces – and it can be then extended or specialized by agents themselves creating new artifacts or replacing existing ones.

The effectiveness of the action model for implementing coordination mechanisms – which was a second main outcome remarked in Section 3 – should be clear from the example shown in Fig. 4. The example shows a `RendezVous` artifact which can be used by N agents to achieve a synchronization point and an agent plan `meeting_plan` in which the artifact is used. From the agent point of view, this is done by simply performing the `ready` action, which is mapped onto the related operation of the artifact. The artifact is programmed so that the operation completes with success only when N agents have executed the same operation. The approach is more efficient compared to any other possible solutions based on pure message passing. For instance, a purely decentralized solution based on an interaction protocol among the N agents would require the exchange of $2(N-1)$ messages and a proper plan – in the agent program – to track and manage the arrival of messages, updating a local belief about the number of agents that reached the meeting point. Conversely, a centralised solution based on an external agent coordinator would require $2N$ messages ($(N-1)^2$ using one of the N agents as the coordinator). In our solution a single action is executed by an agent, so N actions are necessarily in the overall to achieve synchronization.

4.2 Exploiting the perception model

By applying the new perception model, percepts received by an agent who is focussing an artifact are events that concern signals and observable properties updates; such events are used then to automatically update the beliefs in the belief-base of the agent that keep tracks of the current value of the observable properties of the artifact focussed by the agent. By adopting this model we have the guarantee that no states are lost for an agent who is observing the environment (or a part of it). Here we show this in practice by revisiting the shared resource example used in Section 2. In particular the objective of the example is the same, as well as the source code of the agents, but in this case the environment is implemented in CArtAgO by an artifact called `ResourceArtifact`[4]. The artifact – whose source code is reported in Fig. 5 – provides the same functionalities of the Jason environment developed in the previous example. In this case, the underlying perception model ensures that *every change to the*

[4] The example is included in CArtAgO 2.0 distribution, available at `http://cartago.sourceforge.net`

```
// worker source code

times(100).

!run_worker.
+!run_worker : times(X) & X \== 0
  <- acquire_resource;
     !use_resource;
     release_resource;
     -+times(X-1);
     !!run_worker.

-!run_worker : true
  <- .wait(10); !!run_worker.
```

```
// Observer agent source code

last_occurrence(0).
!setup.

+!setup : true
  <- make_artifact("res","ResArtifact",Id);
     focus(Id).

@my_plan [atomic]
+max_use(Occurrence) : last_occurrence(N)
  <- .print("New max usage perceived: ",
            Occurrence);
     !check_state_lost(Occurrence, N);
     -+last_occurrence(Occurrence).

+!check_state_lost(Current,Local): true
  <- X = Local + 1;
     if (Current > X){
       .print("State(s) lost! from ",
              Local," to ",Current)
     }.
```

```
// Resource Artifact implemented in CArtAgO

public class ResArtifact extends Artifact {

  int resourceUse = 0;
  int maxResourceUse;

  void init(){
    defineObsProperty("max_use",0);
    maxResourceUse = 3;
  }

  @OPERATION void acquire_resource(){
    if(resourceUse == maxResourceUse){
      failed("error");
    } else{
      resourceUse++;
      if (resourceUse == maxResourceUse){
        int val = getObsProperty("max_use").intValue();
        updateObsProperty("max_use", val + 1);
      }
    }
  }

  @OPERATION void release_resource(){
    if(resourceUse == 0) {
      failed("error");
    } else {
      resourceUse--;
    }
  }
}
```

Fig. 5. The *resource environment* example introduced in Section 2 implemented here by means of a `ResourceArtifact`. The source code of the workers and of the observer (on the right) is almost the same.

`max_use` observable property is perceived by the `observer` agent, which, differently from the one in Section 2, never prints to standard out any log message about loss of states. Is worth noting that the source code of both `worker` and `observer` agent is almost the same used in the previous example, with minor differences, such as the explicit creation of the `ResourceArtifact` by the `observer` agent.

5 Conclusion

As remarked by the EIS initiative [1], the definition of a general-purpose and standard environment interface is a relevant issue of current APL. This is even more important when developing multi-agent programs that aims at exploiting endogenous environments, as first-class abstraction to encapsulate functionalities. Accordingly, in this paper we discussed the main features of an action and perception model that effectively exploit key properties of endogenous environments, simplifying and making it more efficient agent and environment programming.

Then, we evaluated the approach by implementing it in the new version of CArtAgO, which aims at providing a general-purpose and standard model to conceive, design and program endogenous environments in MAS. In order to achieve this objective, future works include: *(i)* the evaluation of the approach

using other APL besides Jason, starting from GOAL and 2APL; *(ii)* analyzing how the semantics devised in this model and the one proposed in the EIS initiative [1] can be suitably integrated; *(iii)* a formalization of the model and the analysis of its impact on existing agent programming language formalizations.

References

1. T. Behrens, D. Jürgen, and K. V. Hindriks. Towards an environment interface standard for agent-oriented programming. Tech. rep. IfI-09-09. Dept. of Informatics, Clausthal University of Technology.
2. R. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason.* John Wiley & Sons, Ltd, 2007.
3. R. H. Bordini, J. F. Hübner, and R. Vieira. Jason and the golden fleece of agent-oriented programming. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, pages 3–37. Springer-Verlag, 2005.
4. M. Dastani. 2apl: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.
5. J. Ferber and J.-P. Müller. Influences and reaction: a model of situated multi-agent systems. In *Proc. of the 2nd Int. Conf. on Multi-Agent Systems (ICMAS'96).* AAAI, 1996.
6. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.
7. K. V. Hindriks. Programming rational agents in GOAL. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications (2nd volume)*, pages 3–37. Springer-Verlag, 2009.
8. A. Omicini, A. Ricci, and M. Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17 (3), Dec. 2008.
9. A. Omicini, A. Ricci, M. Viroli, C. Castelfranchi, and L. Tummolini. Coordination artifacts: Environment-based coordination for intelligent agents. In *AAMAS'04*, volume 1, pages 286–293, New York, USA, 19–23July 2004. ACM.
10. E. Platon, M. Mamei, N. Sabouret, S. Honiden, and H. V. D. Parunak. Mechanisms for environments in multi-agent systems: Survey and opportunities. *Autonomous Agents and Multi-Agent Systems*, 14(1):31–47, 2007.
11. A. Ricci, M. Piunti, L. D. Acay, R. Bordini, J. Hübner, and M. Dastani. Integrating Artifact-Based Environments with Heterogeneous Agent-Programming Platforms. In *AAMAS'08*, pages 225–232, 2008.
12. A. Ricci, M. Piunti, M. Viroli, and A. Omicini. Environment programming with CArtAgO. In R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Tools and Applications.* Springer, 2009.
13. S. Russell and P. Norvig. *Artificial Intelligence, A Modern Approach.* Prentice Hall, 2003.
14. D. Weyns, A. Omicini, and J. J. Odell. Environment as a first-class abstraction in multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30, Feb. 2007.
15. M. Wooldridge. *An Introduction to Multi-Agent Systems*, chapter Intelligent Agents. John Wiley & Sons, Ltd, 2009.

An Interface for Agent-Environment Interaction

Tristan M. Behrens[1], Koen V. Hindriks[2], Rafael H. Bordini[3], Lars Braubach[4], Mehdi Dastani[5], Jürgen Dix[1], and Jomi F. Hübner[6] Alexander Pokahr[4]

[1] Clausthal University of Technology, Germany {behrens,dix}@in.tu-clausthal.de
[2] Delft University of Technology, The Netherlands k.v.hindriks@tudelft.nl
[3] Federal University of Rio Grande do Sul, Brazil r.bordini@inf.ufrgs.br
[4] Hamburg University, Germany {braubach,pokahr}@informatik.uni-hamburg.de
[5] Utrecht University, Utrecht, The Netherlands mehdi@cs.uu.nl
[6] Federal University of Santa Catarina, Brazil jomi@das.ufsc.br

Categories and subject descriptors: I.2.5 [**Artificial Intelligence**]: Programming Languages and Software; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Intelligent Agents*; I.6.3 [**Simulation and Modeling**]: Applications; I.6.7 [**Simulation Support Systems**]: Environments

General terms: Standardization

Keywords: Agent development techniques, tools and environments, and Case studies and implemented systems

Abstract. Agents act and perceive in shared environments. Although there are many environments for agents – ranging from testbeds to commercial applications – such environments have not been widely used because of the difficulty of interfacing agents with those environments. A more generic approach for connecting agents to environments would be beneficial for several reasons. It would facilitate reuse, comparison, the development of truly heterogeneous agent systems, and increase our understanding of the issues involved in the design of agent-environment interaction. To this end, we have *designed and developed a generic environment interface standard*. Our design has been guided by existing agent programming platforms. These platforms are not only suitable for developing agents but also already provide some support for connecting agents to environments. The interface standard itself is generic, however, and does not commit to any specific platform features. The interface proposal has been implemented and evaluated in a number of agent platforms.

1 Introduction

Agents are situated in environments in which they perceive and act. From an engineering point of view, an issue that repeatedly has to be dealt with is how to connect agents to environments. Sometimes this issue is (partially) solved by the physical sensors and actuators provided (e.g. in the case of a robot). But even if sensor and actuator specifications are available, the design and implementation of

the interaction between the agents and the environment still require substantial effort. This is due in part to the fact that each environment is different but also because the platforms to build agents provide different support for agent-environment interaction.

By now, there exist many interesting environments which range from specialized testbeds for agent systems to industrial applications based on agent technology. In each of these applications, the interaction between agents and environments has to be addressed. This is particularly true in application areas for agent technology such as multi-agent based simulation and the use of agents in (serious) gaming [10,19,20]. In the former, agents need to be connected to computational models of real-world scenarios whereas in the latter agents are used to control virtual characters that are part of a game. The design of agent-environment interaction raises many interesting issues such as who is in control of particular features of the system and what would be the right level of abstraction of the interface that supports the interaction. Technically, there are also many challenges as witnessed by [9] who discuss an interface for connecting agents to the game Unreal Tournament 2004. This gaming environment has been identified as a potentially interesting testbed for multi-agent systems [6]. But without a suitable, generic interface that supports flexible agent-environment interaction such a testbed is unlikely to be widely used.

The availability of many interesting environments for applying agents does not mean that they are easily accessed by agents that are built using different platforms. To the contrary, in practice, it is often the case that agent developers rebuild very similar environments such as grid-like environments from scratch (one well-known toy example is the Wumpus environment [26] of which many implementations exist). Apart from the duplicate work of developing these environments, this also means that dedicated interfaces for agent-environment interaction are built: this makes it difficult to reuse existing environments. Instead, it would be much better to work with an *environment interface standard* which provides all the required functionality for connecting agents to an environment in a standardized way. If environments were developed using such a standard, they could be exchanged freely between agent platforms that support the standard and thus would make already existing environments widely available.

In this paper, we propose an *environment interface standard* that facilitates the sharing and easy exchange of environments for agents. Such a standard will facilitate the reuse of environments between agent platforms; it will support the easy distribution of environments such as the Multi-Agent Contest [13], Unreal Tournament, and many others. There are, however, many other benefits. An environment interface standard will provide a standardized and general approach for designing agent-environment interaction: this is important for the comparison of agent platforms as it would ensure that the same interface is used by each platform. Moreover, a generic interface will support the development of truly heterogeneous MAS, consisting of agents from several platforms. From a more abstract point of view, the design of an interface standard will also increase insight and conceptual understanding of agent-environment interaction.

Our approach is to design an interface that is *as generic as possible*, and that facilitates *reuse as much as possible* from existing interfaces. Clearly, there is a trade-off between these two goals. Our strategy for designing a generic environment interface is (1) to start with what is currently "out there" in existing platforms, and (2) to try to merge this into a generic interface which is sufficiently close to these approaches. As agent-oriented programming platforms seem particularly suitable for developing agents, we have chosen to use four of the more well-known agent programming languages (APLs) as our starting point.

The paper is organized as follows. The design of an environment interface requires a *meta model* of environments, agents, and agent platforms. In Section 2, the principles and requirements such a meta-model should satisfy are identified and the basic components of the model, their interrelations, and the functionalities provided are described. The meta model is used in Section 3 to define the proposed *environment interface standard*, the main contribution of this paper. Section 4 discusses related work and Section 5 evaluates the proposed standard.

2 Requirements and Meta-Model

In this section we will explain the requirements and the meta-model for a general environment interface standard, called EIS.

2.1 Principles

Two of the main motivations for introducing a generic environment interface are to facilitate the easy exchange of environments between agent platforms and to gain a more thorough understanding of the issues related to agent-environment interaction. The environment interface should allow for: (1) wrapping already existing environments, (2) creating new environments by connecting already existing applications, and (3) creating new environments from scratch. To this end, in this section we discuss and present requirements such an interface should satisfy. We do so by introducing various principles the interface should adhere to. We have analyzed the agent-environment support provided by four well-known agent programming languages: 2APL [12], GOAL [15], Jadex [8], and Jason [18]. Based on the principles, we then present a meta-model for an agent-environment interface that is able to provide at least the support for agent-environment interaction already provided by existing agent platforms (Section 2.2).

In order to guide the design of the interface, and to ensure that the interface meets our objectives, we have identified a number of principles we believe a generic environment interface should meet. First, as we aim for a generic interface, the interface should impose as few restrictions on agent platforms and environments as possible. More specifically, we believe that an environment interface should *not* impose: (1) scheduling restrictions on the execution of actions (actions can be scheduled by the agent platform and/or by the environment), (2) restrictions on agent communication or organization structure (communication facilities may be provided by the agent platform as well as by the environment),

(3) restrictions on what is controlled in an environment or how this control is implemented, and (4) restrictions on how various components of the model should be implemented; for example, the interface should allow for different types of agent-environment connection (e.g. TCP/IP, RMI, JNI).

Second, as the interface is aimed at facilitating comparison of agent platforms, a strict separation of concerns is advocated: the interface should not make any assumptions about either the agent platform or the environments such platforms are connected to, except for the type of connection that is established and associated functionalities. In our meta-model, this will be represented by a clear distinction between agents and what we call controllable entities (i.e. "agents' bodies situated in the environment"). Technically, this means the environment interface must abstract from all implementation details concerning both agents as well as environment objects. Instead, the environment interface may only store identifiers to agents and entities and should administrate which agents are associated with which entities (i.e. "who controls which body"). This level of abstraction is required to ensure that no additional implementation effort is required once the agent platform has been adapted.

Finally, as a more technical requirement, the interface should support portability, i.e., the easy exchange of environments from one agent platform to another.

2.2 Meta-Model

We have identified five components that are part of the meta-model on which we base the design of the proposed environment interface. This meta-model is illustrated in Fig. 1, and includes an *environment model*, an *environment interface* that consists of an *environment management system* and an *environment interface* component, an *agent platform* and *agents*.

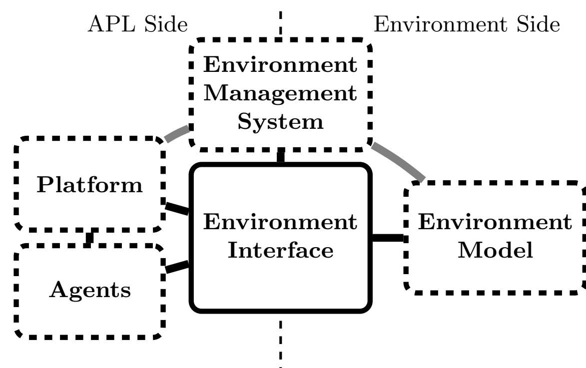

Fig. 1. The components of our environment meta-model.

Our *environment model* assumes the presence of a specific kind of entity. [7] defines an entity as "any object or component that requires explicit representation in the model." In the context of agent-environment interaction, the entities

that we are interested in may be controlled by an agent. This means that the behavior generated by the entity can be controlled by an agent if the agent is properly connected to the entity. It is the task of the interface to establish such a connection. Entities in an environment that can be so controlled are called *controllable entities*.

Controllable entities facilitate the connection between the agents running on an agent platform and an environment by providing identifiers, *effecting* capabilities, and *sensory* capabilities to agents. An agent's identifier allows the environment to send percepts or events to agents by means of the interface. Moreover, the effecting and sensory capabilities specified by controllable entities allow the environment to contextualize an agent's action repertoire, the actions' effects, and which part of the environment can be sensed, thus establishing the *situatedness* of the agents.

The objective of defining an environment interface standard is to provide a generic approach for connecting *agents* to environments. Agents may refer to almost any kind of software entity but the stance taken here is that agents are able to perform actions in the environment, sense the state of the environment and process such sensorial input, and receive and process events that are generated by the environment. We use the following very generic definition taken from [26] that includes precisely these two aspects: *An agent is anything that can be viewed as* ***perceiving*** *its* ***environment*** *through sensors and* ***acting*** *upon that* ***environment*** *through effectors.* We do not intend to restrict our proposal to any specific kind of agent program, although we are primarily motivated by existing agent-oriented programming languages.

An *agent platform* is the infrastructure that facilitates the instantiation and execution of individual agents. It is also assumed to facilitate connecting agents with environments and associating agents to controllable entities by means of the environment interface functionality. Other than that, nothing else is assumed about an agent platform.

The *environment interface* consists of two components: the *agent-environment interaction* component and the *environment management* component. The agent-environment interaction component manages the mapping and interaction between individual agents and the agent platform on one hand, and the environment and controllable entities on the other hand. The interaction between an agent platform and the agent-environment interaction component allows agents to act in an environment, sense its state, and receive events from it. We allow two ways of sensing: (1) active sensing through specific sense actions, and (2) passive sensing through a generic sense action embedded in the control cycle of the agents. Using the agent-environment interaction component, the platform can process different types of actions by calling different methods of this component and possibly wait for the return values which are subsequently passed on to the platform. The values returned can be either success/failure notifications or sense information if actions were (passive or active) sense actions. The environment interface can also interact with a platform by sending an event to

a specified agent. The platform is then responsible to pass the event on to the specified agent (e.g., by adding the event to the agent's event base).

3 The Proposed Interface

In this section, we explain our ideas for a generic environment interface. First, we define an *interface intermediate language* that facilitates data-exchange (percepts, actions, events) between different components. Second, we assume a functional point-of-view of the interface architecture. The interface provides functions for: (1) attaching, detaching, and notifying observers (software design pattern); (2) registering and unregistering agents; (3) adding and removing entities; (4) managing the agents-entities-relation; (5) performing actions and retrieving percepts; and (6) managing the environment.

3.1 Running Example: Multi-Agent Contest

The Multi-Agent Programming Contest 2010 tournament consists of a series of simulations. In each simulation two teams of agents compete in a grid-like world. There are virtual cowboys that can be controlled by agents. Agents have access to incomplete information, because the cowboys have a fixed sensor-range. Acting means moving a cowboy to a neighboring cell on the grid. There are no further actions. The environment contains obstacles: some cells can be blocked and thus are unreachable. The grid is also populated by virtual cows, that behave according to a simple flocking-algorithm. To win a simulation, an agent team has to herd more cows, and take them to their own corral, than the opponent team. The simulation proceeds through discrete time steps. In each step, agents can perceive, have a fixed amount of time to deliberate, and are then allowed to act. After a number of steps the simulation is finished. The tournament is run by the MASSim-server, which schedules and runs simulations. Agents are supposed to connect to the server as clients. Communication between clients and server is facilitated by exchanging XML-messages via the TCP/IP protocol.

3.2 Interface Intermediate Language

An important design decision has been to define, as part of the environment interface, a convention for representing actions and percepts. This convention is called the *interface intermediate language*, and supports the exchange of percepts and actions from/to environments. A conventional representation for actions and percepts is required to be able to meet the second principle aimed at facilitating comparison of platforms and the fourth principle that aims at easy exchange of environments and portability. To meet these principles, the interface should be agnostic to any implementation details of either agent platform or environment; this can be achieved by an abstract intermediate language. The convention proposed here, however, imposes almost no restrictions (which is in line with our first principle of generality).

The language consists of: (1) *data containers* (e.g. actions and percepts), and (2) *parameters* for those containers. Parameters are *identifiers* and *numerals* (both represent constant values), *functions* over parameters, and *lists* of parameters. Data containers are: *actions* that are performed by agents, *results* of such actions, and *percepts* that are received by agents.

Here is an example for a set of percepts that informs an agent about the beginning of a simulation, including the position of the corral, the size of the grid, the visual range of the agents, the name of the opponent team and the number of steps of the simulation:

```
corral(0,0,20,20)      grid(100,100)   id(1)
opponent(uglydozen)    lineOfSight(8)  steps(1400)
```

The action that establishes a connection to a server at a given location is `connect(agentcontest1,goodbadugly1,hh564kh)`, where `agentcontest1` is the hostname, `goodbadugly1` is the username, and `hh564kh` is the password. An example for an action that contains a list and functions, but is not related to the agent contest is `followPath(entity1,[pos(1.0,0.0),pos(1.0,1.0)])`; this is a high-level action that makes an entity follow a path.

3.3 Functional Point-of-View

What exactly is the correspondence between an environment-interface and the components (platform, agents, etc.)? We allow for a two-way connection via *interactions* that are performed by the components and *notifications* that are performed by the environment-interface.

Interactions are facilitated by function calls to the environment-interface that can yield a return value. For notifications we employ the *observer design pattern* (call-back methods, known as *listeners* in Java). The observer pattern defines that a *subject* maintains a list of *observers*. The subject informs the observers of any state change by calling one of their methods. The observer pattern is usually employed when a state-change in one object requires changes in another one. This is the reason why we made that choice. The subject in the observer pattern usually provides functionality for *attaching* and *detaching* observers, and for *notifying* all attached observers. The observer, on the other hand, defines an *updating* interface to receive update notifications from the subject.

We allow for both interactions and notifications, because this approach is the least restrictive one. This clearly corresponds to the notions of *polling* (an agent performs an action to query the state of the environment) and *interrupts* (the environment sends percepts to the agents as in the *AgentContest* example).

Agents and Entities: We make three assumptions: (1) there is a set of agents on the agent platform side (we do not know anything about them), (2) there is a set of controllable entities on the environments side (again we do not know anything about them), and (3) agents can control entities through the environment-interface. An important design decision that we had to make is to store in the environment-interface only identifiers to the agents, identifiers to the entities, and a mapping between these two sets. The reason for that decision is,

as mentioned before, that we do not assume anything about the agent platform side or the environment side. Fig. 2 shows the agents-entities relation. The agents live on the agent platform side, they are known by the environment-interface through their identifiers. The entities live on the environment-side, and they are also known by their identifiers. The agents-entities relation is stored as a mapping between both sets of identifiers. In the *AgentContest*, each cowboy is a controllable entity. Cows are entities as well but they are not controllable. Each agent can control only a single cowboy.

In general, we allow the agents-entities relation to be arbitrary. For example, we also allow for one agent to be associated with several entities. This would be useful when using the agents&artifacts meta-model [23] to provide means for agent-coordination through the environment. An artifact would be an entity that can be controlled by several agents. Agents would perceive the state of the artifact and can act so as to change it.

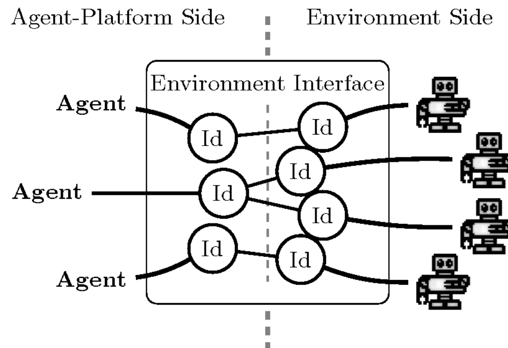

Fig. 2. The agents-entities relation.

Attaching, Detaching, and Notifying Observers: There are two directions for exchanging data between components and environment interfaces. One is via environment observers, which inform observers about changes in the environment or the environment interface. The second is via agent observers, which send percepts to agents. In order to facilitate sending events (i.e. percepts as notifications and environment events), the interface provides functions that allow for attaching and detaching observers, and for notifying components connected via observers. Listeners are useful when connecting to the *AgentContest* environment, since it is the simulator that actively provides agents with percepts.

Registering and Unregistering Agents: This step is the first to facilitate the interaction between agents and environments and establishing the agents' situatedness. It is necessary for the internal connection between agents and entities. The interface provides two methods: one for registering (`registerAgent`), and one for unregistering an agent (`unregisterAgent`). We note that only identifiers representing the agents are stored and managed by the interface.

Adding and Removing entities: Entities are added and removed in a similar fashion to agents. Again identifiers representing entities are stored instead of the entities themselves. There are two methods: the first (`addEntity`) adds, and the second one (`deleteEntity`) removes an entity. Again this is necessary to facilitate the connection between agents and entities. Once an entity is added or removed, any observing components (platform and/or agents depending on the design of the platform) are notified about the respective events. This is done in order to allow components to react to changes in the set of entities in an appropriate manner.

Managing the Agents-Entities Relation: Associating an agent with one or several entities is the second and final step of establishing the situatedness of agents by connecting them to entities that provide effectory and sensorial capabilities. The agents-entities relation is manipulated by means of three methods. The first method (called `associateEntity`) associates an agent with an entity, the second one (`freeEntity`) frees an entity from the relation, and the third one (`freeAgent`), frees an agent. This can be done by the interface internally and by other components that have access to it as well. Restrictions on the structure of the relation can be established by the interface. In the *AgentContest*, for example, one agent is supposed to control at most one virtual cowboy.

Performing Actions and Retrieving Percepts: The agents-entities relation is a connection between agents and the sensors and effectors of the associated entities. We establish two directions of information flow. Each direction corresponds to a typical step in common agent deliberation cycles. We have facilitated the management of the two directions of flow by following a unified approach whereby two methods are provided by the interface. The first one (`performAction`) allows an agent to act in the environment through the effectors of its associated entities. The second method (`getAllPercepts`) allows an agent to sense the state of the environment through the sensors of the associated entities. In the "cows and cowboys" scenario, nine actions are available. One for connecting to the server at a given IP address with valid username and password, and the other eight for moving the cowboy in each possible direction. The method `getAllPercepts` retrieves the last percept sent by the server.

Managing the Environment: Although different environments provide different support to manage the initialization, configuration, and execution of the environment itself, it is useful to include support for environment management in the environment interface. For example, it is often useful to be able to "freeze" a running MAS simultaneously with the environment to which the MAS is connected by means of pause functionalities provided by the platform and the environment. As there is no common functionality supported by each and every environment, we have chosen to provide support for environment management by introducing a *convention* for labeling a set of *environment commands* and *environment events*. The commands that are part of the proposed environment management convention include *starting*, *pausing*, *initializing*, *resetting*, and *killing* the environment.

3.4 Implementation Details

The goal of developing an environment interface standard is to facilitate the easy exchange of environments. The interface would reduce the implementation effort of connecting agent platforms to environments. Of course, the effort of connecting to the environment through an environment interface should not substantially increase the effort needed for directly connecting agents to an environment. Below, we report on the experience we gained with adapting four agent platforms so that they support the environment interface as well as the experience gained with two environments that were adapted to support the environment interface.

In order to create an environment interface for a given environment, dedicated code that is specific to the environment is necessary. To that end, a particular Java interface has to be implemented. That interface enforces the functional contract introduced in subsection 3.3. Alternatively, the developer can inherit from a class that contains a default implementation for all of the contract's methods. Whatever path the developers follow, they need to establish a connection to the environment.

Supported Agent Platforms To evaluate the ease of use and generality of the developed EIS concepts and components, we have connected four different APLs to example environments developed with the EIS. For 2APL, GOAL, JADEX, and JASON, a connection has been established with less than one day of coding effort each.

2APL proved to be compatible with EIS. In order to establish a connection a two-way converter for the interface intermediate language had to be developed. Furthermore, the environment loading mechanism of 2APL had to be replaced with the environment-interface loading mechanism provided by EIS. Percepts sent by EIS using the observer functionality are translated into 2APL events and handed over to the event-handling mechanism of the interpreter. Finally, special external actions have been added to facilitate the manipulation of the agents-entities relationship: (1) retrieving all entities, (2) retrieving all free entities, (3) associating with one or several entities, and (4) disassociating with one or several entities.

The original environment interface of GOAL did not fit with everything provided by the environment interface. It nevertheless proved quite easy to connect the interface to GOAL as most functionality provided by the interface is straightforwardly matched to that provided by the GOAL agent platform. Similar to 2APL, a two-way converter for the interface intermediate language had to be developed with little effort required. There were no percepts as notifications (like events in 2APL), prior to the adaptation to EIS. GOAL only allowed for retrieving all percepts in a distinct step of the deliberation cycle. Percepts as notifications are now collected and processed together in the step where all percepts are processed. Also, the MAS file specification of GOAL has been extended. Now one can use launch rules to connect specific agents with specific entities. This allows for instantiating agents even during runtime.

For connecting JADEX agents to EIS, it is sufficient to make all agents of one application aware of the concrete EIS object, implementing the current environment. In order to do this in a systematic way, the JADEX concept of *space* was used. A space may represent an arbitrary underlying structure of a MAS that is known by all agents. To support the EIS, a special `EISSpace` has been provided, which implements the required interfacing code for connecting to an EIS-based environment. Therefore, the participation in such an environment can now simply be specified in the JADEX application descriptor (".application.xml"). When such a defined application is started, the initial agents as well as the EIS environment will be created. Agents can then access EIS by fetching the corresponding space from their application context and use the EIS Java API directly for, e.g., performing actions or retrieving percepts.

JASON's integration with EIS was straightforward since almost all concepts used in the EIS are also available in Jason. The integration consists essentially of: (1) the conversion of data types, and (2) the development of a class that adapts EIS environments to JASON environments. In regards to (1), all EIS data types have an equivalent in JASON. Although some data types in JASON (e.g., Strings) do not have a corresponding type in EIS, they can be translated to EIS identifiers. In regards to (2), the adaptor is a normal JASON Environment class extension that delegates perception and action to the EIS. The adaptor class is also responsible for registering the agents with the EIS as they join a JASON multi-agent system and wake them up when the environment changes (using the observer mechanism available in EIS). From all the concepts used in EIS, only that of "entities" is not supported by JASON as all actions and perceptions are relative to an agent and the overall environment rather than a particular entity therein. For sensing, the chosen solution was to add annotations to percepts that indicate the entity of origin. For actions, in case the agent is associated with exactly one entity, the action is simply dispatched to that entity. Otherwise, a special action that receives the relevant entity as a parameter must be used.

Implemented Environments The environment interface comes with several very simple examples of environments for illustrative purposes. These examples are mainly provided for clarifying some of the basic concepts related to the interface. We briefly discuss here two EIS-enabled environments, that may be used by any agent platform that supports EIS.

The *elevator environment* is a good example of an environment that was not built specifically with agents in mind, and is available from [1]. The environment is a simulator of arbitrary multi-elevator environments where the elevators are the controllable entities and the people using the elevators are controlled by the simulator. It comes with a graphical user interface (GUI) and a set of tools for statistical analysis. The environment had been originally adapted for the GOAL platform. The additional effort required to re-interface that environment to EIS was very little. The main issue was the event handling related to the initial creation of elevators, a functionality provided and supported by the environment interface which required some additional effort for adapting the environment to

provide such events. The environment provides actions that take time (durative actions) instead of discrete one-step actions, which illustrates that the interface does not impose any restrictions on the types of actions that are supported. Similarly, elevators only perceive certain events but not, for example, whether buttons are pressed in other elevators. We have successfully used the elevator environment with GOAL and 2APL.

Connecting to the MASSim-server turned out to be easy. As already mentioned, the entities in the *AgentContest*-environment are cowboys that herd cows. From the implementation point-of-view each connection to an entity is a TCP/IP connection. Acting is facilitated by wrapping the respective action into an XML-message and sending it to the server. Perceiving is done by receiving XML-messages from the server and notifying possible agent-listeners. Furthermore, for the sake of convenience, percepts are stored internally for a possible active retrieval. Much effort had to be invested in mappings from the interface intermediate language to the XML-protocol of the *AgentContest* and vice versa. We have shown that the interface does indeed not pose any restrictions on the connection between itself and environments.

Finally, it is worth mentioning that an interface to Unreal Tournament 2004 [16] is under developmentGrown out of the need for a more extensive evaluation of the application of logic-based BDI agents to challenging, dynamic, and potentially real-time environments, this EIS interface might help putting agent programming platforms to the test.

Evaluation Summary The relative ease with which the interface has been connected to four agent platforms and various environments already indicates that the interface has been designed at the right abstraction level for agent-environment interaction. The four agent platforms differ in various dimensions, regarding, for example, the functionality provided for handling percepts and actions (is the platform more logic-oriented or Java-based?) and how environments were connected to these platforms before using the interface. The environment interface nevertheless could be connected to each of the platforms easily, thus providing evidence of its generality and as well. Of course, we need more agent platforms to use the environment interface, and we have invited other platform developers to do so, but we do not expect this will pose any fundamental new issues. Initial experience with various environments has also shown that little to no restrictions are imposed on the types of environments that can be connected to an agent platform using the interface. The interface, for example, can support both real-time or turn-based environments, as well as environments that differ in other respects. Although we have mainly discussed software environments, there is no principled restriction imposed by EIS that would make it only applicable to such environments. It has been shown already in the past that it is possible to connect agent platforms to embedded platforms such as robots. EIS just provides another, more principled approach for doing so. In fact, it is planned to use EIS to connect to a robotic platform in the near future.

4 Related Work

The EIS was designed as a building block for an agent application, providing a standardized way of interfacing the agents with environmental components. In the context of agent applications, at least the following forms of environments can be distinguished: (1) environments in agent-based simulation models, (2) virtual environments such as testbeds or computer games, (3) real application components such as enterprise information systems, and (4) coordination infrastructures.

Agent-based simulation models can be used for performing experiments and analyzing the obtained result data. Agent simulation toolkits are specifically designed for this purpose and often employ custom agent models (e.g. simple task-based agents) and a proprietary form of defining the environment behavior. Usually, there is a tight coupling between agents and the environment that is designed to support these toolkit-specific models. Therefore, simulation toolkits are closed in the sense that they do not support (and are not meant to) connecting external agents to simulated environments or simulated agents to external environments.

Agent programming testbeds and contests, such as TAC, [5], RoboCup, [4] and the Multi Agent Contest [3], are specifically designed to offer open interfaces for connecting different types of agents to the provided test environment. Moreover, some network-based computer games with remote playing capabilities (e.g. Unreal Tournament) offer interfaces for controlling entities in the game environment which have been adapted to connect to software agents instead of human players [9]. All of these interfaces are quite specific with regards to the testbed or game they were created for, and therefore agent platform developers have to repeat the implementation effort of connecting their agents to each of these interfaces.

To connect agents to an environment composed of real application components, different options are available. Application-centered approaches would directly use available component interfaces or domain specific standards (such as HL7 in the healthcare domain) for the connection. Depending on the severity of the "impedance mismatch" between the component interface and the agent platform, this can become quite laborious and additionally has to be repeated for each platform and each application. Agent-centered approaches try to "agentify" the environment components, leading to a more seamless and straightforward connection. For example WSIG (Jade) [2] is an infrastructure that allows agents to interact with web services as if they were agents and vice-versa.

One well known approach for coordinating agents is by using blackboard approaches, which offer agents a possibility to decouple their interactions in terms of time and potential receivers. Besides passive blackboards acting as information stores only, also more advanced tuple spaces such as ReSeCT [22] have been devised with which one can also capture domain logic in terms of rules. The Open Agent Architecture (OAA) [11] is another form of coordination environment, in which the cooperation among agents and also humans is facilitated by automatic task delegation and execution. In contrast to EIS, these approaches focus on in-

formation exchange and problem solving and do not tackle the question of how environments could be generically interfaced.

Organizational or institutional approaches such as Islander [14] and MOISE [17] regulate agent behavior at high-level allowing designers and/or agents to define, monitor, and enforce certain kinds of organizational constraints (e.g. norms and group membership). The latest platform for MOISE is founded on the notion of organizational environment where agents can perceive and act on their organization. This kind of environment can also contain artifacts specially developed to enforce some norms (e.g. a surgical room's door that forbids agents to enter if they do not play the role of doctor). Other approaches affect more directly agent behavior, for example biologically inspired approaches such as pheromone-based techniques to guide agent movement. While these approaches make use of the notion of environment, they are quite domain specific and do not allow for arbitrary environment development. In contrast, the A&A model [23] has been proposed as a generic paradigm for modeling environments. In the A&A paradigm, an application is composed of agents as well as so called artifacts. While the model makes no restricting assumptions with respect to the agents, the interface and operation of an artifact is intentionally quite rigidly defined. An implementation of the A&A model is available in form of the distributed middleware infrastructure CArtAgO [25].

We see EIS not as a competitor, but rather as a desirable complement to the above mentioned approaches. For example, one possible use of the EIS standard is reducing the required implementation effort for connecting agent to, say, virtual environments, as once an EIS-based interface has been developed for a contest or game, it can easily be reused by different agent platforms. Unlike FIPA-compliant approaches such as the WSIG, the focus of the EIS is providing a lean interface, i.e., when FIPA-compliant communication is not necessary, the EIS allows achieving similar openness and portability with much less effort. In particular, we see much potential in a combination of EIS and CArtAgO. Currently, there are specific bridges available for connecting agent platforms such as JADEX, JASON and 2APL to CArtAgO [24]. Implementing an EIS bridge for CArtAgO could lead to a universal implementation that could be used to connect CArtAgO to any agent platform (if it is already EIS-enabled). In general, the EIS standard will facilitate connecting any agent platform to all sorts of environments (A&A based as well as others).

5 Conclusion

The design and implementation of our proposal for an environment interface standard (see [**?**] for a more detailed exposition and more technical details) is motivated by the fact that it has been difficult to connect arbitrary agent platforms to many of the available environments. The design of the interface provides additional insight into the general problem of agent-environment interaction. At a conceptual level, the development of the environment interface has yielded insight, for example, into some of the distinguishing features of existing agent

platforms. For example, where some platforms expect events initiated by the environment other platforms are based on a polling model for retrieving percepts.

The initial results of applying the interface to various agent platforms and environments have been very encouraging: they demonstrate the generality and usability of our interface. The environment interface standard allows the portability and reuse of application and testing environments across existing and newly developed agent platforms. Furthermore, it provides a basis for heterogeneous agent applications composed of agents implemented in different agent platforms. The experience so far has also shown that connecting to and using the interface requires minimal effort and can be implemented easily.

Although the environment interface proposed here provides a solid basis for agent-environment interaction, there are some topics that require additional work. One of these topics involves the environment management system which has only been partly supported by most agent platforms; it facilitates combinations of agent platform and environment functionalities such as combined resetting of MAS and environment, but this requires additional investigation. We also need to gain more experience with the dynamic addition and removal of entities and the handling of such events by platforms. Related to the previous point, there is the issue of managing various types of entities. For example, how can the interface be extended to support the identification of these different types? Finally, we need to get more agent platforms, including platforms from multi-agent based simulation and other areas, involved and support the environment interface to establish our proposal as a genuine (de facto) standard.

References

1. Elevator simulator homepage. `http://sourceforge.net/projects/elevatorsim/`.
2. Java Agent DEvelopment Framework homepage. `http://jade.tilab.com/`.
3. Multi Agent Contest homepage. `http://www.multiagentcontest.org/`.
4. RoboCup official site. `http://www.robocup.org/`.
5. Trading Agent Competition homepage. `http://www.sics.se/tac/`.
6. R. Adobbati, A. Marshall, A. Scholer, S. Tejada, G. Kaminka, S. Schaffer, and C. Sollitto. Gamebots: A 3d virtual world test-bed for multi-agent research. In *Proceedings of the 2nd Int. Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, 2001.
7. J. Banks, J. S. Carson, B. L. Nelson, and D. M. Nicol. *Discrete-Event System Simulation*. Prentice Hall, 2009.
8. T. M. Behrens, J. Dix, and K. V. Hindriks. Towards an environment interface standard for agent-oriented programming. Technical Report IfI-09-09, Clausthal University of Technology, Sept. 2009.
9. R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.
10. O. Burkert, R. Kadlec, J. Gemrot, M. Bda, J. Havlcek, M. Drfler, and C. Brom. Towards fast prototyping of IVAs behavior: Pogamut 2. In *Proceedings of 7th International Conference on Inteligent Virtual Humans*, 2007.

11. M. Buro. Call for AI Research in RTS Games. In *AAAI-04 AI in Games Workshop*, 2004.
12. A. Cheyer and D. Martin. The open agent architecture. *Journal of Autonomous Agents and Multi-Agent Systems*, 4(1):143–148, March 2001. OAA.
13. M. Dastani. 2apl: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.
14. M. Dastani, J. Dix, and P. Novák. Agent Contest Competition - 3rd edition. In M. Dastani, A. Ricci, A. El Fallah Seghrouchni, and M. Winikoff, editors, *Proceedings of ProMAS '07, Revised Selected and Invited Papers*. Springer, 2008.
15. M. Esteva, D. de la Cruz, and C. Sierra. Islander: an electronic institutions editor. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 1045–1052, New York, NY, USA, 2002. ACM.
16. K. V. Hindriks and T. Roberti. Goal as a planning formalism. In *MATES 2009 Proceedings*, pages 29–40, 2009.
17. K. V. Hindriks, B. van Riemsdijk, T. Behrens, R. Korstanje, N. Kraaijenbrink, W. Pasman, , and L. de Rijk and. Unreal GOAL bots. In *Preproceedings of The AAMAS-2010 Workshop on Agents for Games and Simulations*, to appear.
18. J. F. Hübner, O. Boissier, R. Kitio, and A. Ricci. Instrumenting multi-agent organisations with organisational artifacts and agents: "giving the organisational power back to the agents". *Journal of Autonomous Agents and Multi-Agent Systems*, 2009.
19. B. Lars, P. Alexander, and L. Winfried. Jadex: A BDI-agent system combining middleware and reasoning. In V. R. Unland, M. Klusch, and M. Calisti, editors, *Software agent-based applications, platforms and development kits*, 2005.
20. R. Z. Mili and R. Steiner. Modeling Agent-Environment Interactions in Adaptive MAS. In *EEMMAS 2007*, pages 135–147. Springer-Verlag, 2008.
21. J. Müller. Towards a Formal Semantics of Event-Based Multi-Agent Simulations. In *MABS 2008*, pages 110–126. Springer-Verlag, 2009.
22. A. Omicini. Formal respect in the a&a perspective. *Electron. Notes Theor. Comput. Sci.*, 175(2):97–117, 2007.
23. A. Omicini, A. Ricci, and M. Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3):432–456, 2008.
24. A. Ricci, M. Piunti, L. D. Acay, R. Bordini, J. Hübner, and M. Dastani. Integrating artifact-based environments with heterogeneous agent-programming platforms. In *7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-08)*, pages 225–232. IFAAMAS, 2008.
25. A. Ricci, M. Viroli, and A. Omicini. CArtAgO: A framework for prototyping artifact-based environments in MAS. In D. Weyns, H. V. D. Parunak, and F. Michel, editors, *Environments for MultiAgent Systems III*, pages 67–86. Springer, 2007.
26. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2nd edition, 2003.

Evaluating Agent-Oriented Programs: Towards Multi-Paradigm Metrics

Howell R. Jordan, Rem Collier

Lero @ University College Dublin, Ireland

Abstract. Metrics are increasingly seen as important tools for software engineering and quantitative research, but little attention has so far been devoted to metrics for agent programming languages. This paper presents the first steps towards multi-paradigm structural metrics, which can be applied seamlessly to both agents and the object-oriented environments in which they are situated - thus enabling the designs of complete multi-agent systems to be quantitatively evaluated. Concrete paradigm-independent metrics for coupling and cohesion are proposed, and their use is demonstrated on an example Jason program, written in AgentSpeak and Java.

Categories and subject descriptors: D.2.8 [**Software Engineering**]: Metrics—*Product Metrics*; I.2.5 [**Artificial Intelligence**]: Programming Languages and Software

General terms: Measurement, Design, Languages

Keywords: Design metrics, structural metrics, agent programming languages

1 Introduction

Software design metrics, or structural metrics, are an important part of the professional software engineer's toolkit. Beyond their traditional roles in managerial oversight, metrics are increasingly used in iterative development processes to quickly highlight areas which may be vulnerable to defects or resistant to future change [20]. Metrics for object-oriented programming are well-established, and automated collection tools for the best-known metrics suites [9][21] are available for several popular object-oriented programming languages (OOPL).

Agent orientation is an emerging paradigm for software construction, in which software is composed of autonomous, proactive agents situated in some environment. Like other high-level software abstractions - such as components and aspects - agents complement, rather than replace, existing object technology. Agent-oriented software is often implemented using objects, for example by building directly on the popular JADE platform [4], in which case it can be evaluated directly using existing metrics (see for example [14]).

Many researchers have argued that the benefits of agent orientation are best realised using a dedicated agent programming language (APL), thus "fixing the

state of the modules to consist of components such as beliefs, capabilities, and decisions, each of which enjoys a precisely defined syntax" [30]. However, the performance cost of agents is much greater than that of passive objects, and attempting to write an industrial-strength 'pure agent' program has become a recognised pitfall of agent-oriented software development [35]. Many of the current generation of agent programming languages [6] resolve this dilemma by deferring the environment implementation [34] and any lower-level agent processing activities to object technologies such as Java.

To our knowledge, no metrics for agent programming languages have yet been proposed. Furthermore, if software designs are to be evaluated quantitatively, consistently, and comparably across the APL-OOPL divide, there is a clear need for a single metrics suite that is applicable to both domains. But let's not stop there. Inspired by recent research which integrates agents with software components [12], we expect the path towards wider adoption of agent programming languages to be through their integration with other paradigms. In this paper, we present the first steps towards a metrics suite which could, in principle, be used to evaluate software designs expressed in many different text-based programming languages.

The structure of this paper is as follows. In the next section, we discuss how the application of structural metrics to agent programming languages could benefit both practice and research. Section 3 outlines some of the wide literature of related work, and section 4 introduces an example Jason program. In section 5 we propose two structural metrics for agent programming, and apply them to the motivating example. Finally, we conclude by offering some tentative advice to the creators of agent programming languages, and some suggestions for future work in this area.

2 Why Metrics?

Metrics are used in software engineering to measure the quality of a software process or product. In this paper we focus on the product. A software product consists of many linked artifacts, such as code, tests, and documentation; here we focus specifically on the product's design, and the design information contained implicitly in its source code.

2.1 Measures of product quality

Software product quality is typically defined as a combination of factors, characteristics, or attributes [17] [15]. The primary quality factors are often given as functionality, reliability, usability, efficiency, maintainability, and portability. These factors are usually structured as a hierarchy, whereby each primary factor is itself defined as a combination of subfactors.

The depth and complexity of this hierarchy presents a measurement challenge. For a quantity to be measurable it must first be completely defined; unfortunately, there is no universally-agreed definition of software quality. The

relative influence of quality attributes is also highly sensitive to context. For example, in mission-critical applications, reliability is obviously dominant; yet in others, reliability need be no better than 'good enough', and other quality attributes assume greater importance [2]. Attempts to combine multiple metrics into a general-purpose quality measure are therefore fraught with difficulty, and most software product metrics aim to measure a single, specific quality attribute [18].

Maintainability metrics are an important general software engineering topic [25]. However, in this paper, we focus on maintainability as an interesting potential benefit of agent oriented software engineering. Agent programming is thought to aid the design and development of complex software, chiefly by enabling the developer to take full advantage of the intentional stance [30]; but little is currently known about the effects of agent orientation on maintainability.

Maintainability consists of at least three major subfactors: algorithmic complexity, structural or design complexity, and size. Efforts to define multi-paradigm size and algorithmic complexity measures are already well advanced (see section 3), and the rest of this paper will focus on structural metrics. Structural complexity is itself a compound attribute, and among its subfactors, coupling and cohesion are thought to be dominant [11].

We take the view that structural metrics essentially predict how difficult it will be to modify a system in future; and thus they indirectly predict the likelihood of bugs, both in the present and in the immediate aftermath of any future change. In object-oriented systems, this view is supported by a large body of experimental evidence [32]. If the same is true for agents, the development of structural metrics for agent programming languages will help professional software engineers to deliver agent-oriented software of higher quality.

2.2 Tools for software research

Aside from their importance to professional software engineers, product metrics have an important role to play in software research.

Metrics can be used as powerful tools for technology comparison, as demonstrated by Garcia et al. [14]. The research method employed in this study is based on the 'Goal Question Metric' approach [3] and can be summarised roughly as follows: implement functionally-identical solutions to a given problem using two or more different technologies; then compare those solutions using a suitable metric. If suitable multi-paradigm metrics were available, this method could be employed much more widely.

Studies of the above type could also help facilitate adoption of agent technologies by industry. It is widely thought that agent technologies provide greater benefits when used in certain application areas, but little experimental data is currently available to support these opinions. We argue that, given more quantitative evidence of the maintainability of agent oriented solutions, some of the business risks of agent adoption would be mitigated.

Finally, we hope that a focus on maintainability metrics - and on the related topics of software evolution, refactoring, coupling, and cohesion - will lead to

innovations in agent programming language design. In section 6, we tentatively suggest some additions to the AgentSpeak language that might improve the maintainability of AgentSpeak code, based on experiences gained in this study.

3 Related Work

We have been unable to identify any metrics specifically for agent programming languages. Aside from the large literature on object oriented metrics [15], structural metrics have been proposed for rule-based programs [27], concurrent logic programs [36], and knowledge based systems [19]. More recently, several metrics suites for aspect-oriented programming have also been devised [23].

We now turn our attention to other work in the field of multi-paradigm metrics.

Sipos et al. have proposed a multi-paradigm metric for algorithmic (as opposed to structural) complexity [31]. This metric is directly related to the number of independent paths through a given program, and high values therefore have negative implications for program testability. Given the importance of testing (and automated testing in particular) to refactoring and software evolution, we believe this metric is complimentary to ours.

Allen et al. model a program as a hypergraph, from which metrics for program size, algorithmic complexity, and coupling, can be derived [1]. Each metric is defined in detail using arguments from information theory; but no paradigm-independent method for extracting the hypergraph model from source code is offered. The hypergraph model is also insensitive to the strength and directionality of connections between nodes.

The PIMETA approach [7] leads to multi-paradigm structural metrics like ours, but it is restricted by several practical difficulties. PIMETA requires a detailed meta-model to be instantiated for each evaluation, and as the authors note, for many real programs the resulting models are large and difficult to visualise. No precise definition of coupling between PIMETA abstractions is offered, with coupling apparently defined on a case-by-case basis; it is therefore not clear how the meta-model instantiation process might be automated.

The 'separation of concerns' metrics suite of Sant'Anna et al. [29] is also, in part, applicable to agent programming languages. Though it is intended for use in the component-based and aspect-oriented paradigms, some of its metrics (Concern Diffusion over Architectural Components, Component-level Interlacing Between Concerns, Afferent Coupling Between Components, Efferent Coupling Between Components, and Lack of Concern-based Cohesion) could also be applied to agents. However the automated collection of these measures would be problematic, since the concepts of 'architectural concerns' and 'coupling between components' are not precisely defined. The remaining metrics in the suite make use of abstractions not defined for agents, such as distinct interfaces and operations.

4 Motivating Example

We illustrate the need for multi-paradigm software maintainability metrics with a simple example. Figure 4 shows a grid based on the 'vacuum world' of Russell and Norvig, populated by four cleaning robots. The environment, implemented in Java, allows each vacuum cleaner to move north, south, east, and west; to sense its immediate surroundings with a small field of vision; and to clean its current square. The simulation is further enriched by fixed obstacles (shown in a darker colour) and realistic robot movement.

Fig. 1. A grid world of robotic vacuum cleaners and brightly-coloured dust.

In this environment, we pose a simple problem: to clean the grid of dust, as quickly as possible. Without excluding other technologies, this problem's characteristics point towards an agent-oriented solution: the environment is dynamic, with partial, local visibility; and its four independent robots suggest at least four concurrent processes. For this example, we have chosen to implement a solution using Jason, an implementation of the AgentSpeak agent programming language.

Complete source code for the example can be downloaded from `http://www.agentfactory.com`.

4.1 Solution architecture

Our example solution consists of two types of agent: 'vacAgent' and 'bossAgent'. Four instances of vacAgent directly control each of the four vacuum robots. Only one bossAgent is instantiated, and is not situated in the environment; instead, it receives reports from the other agents about the world's state, from which it builds a mental map of the environment, and directs the exploration and cleaning activities of the vacAgents. For simplicity, we consider the environment

as a third-party library with a stable interface - in other words, the actuators and perceptors are not part of our 'solution', and are guaranteed not to change. Each vacAgent relies on object-oriented 'internal actions' to implement some of its functionality, as shown in Figure 4.1.

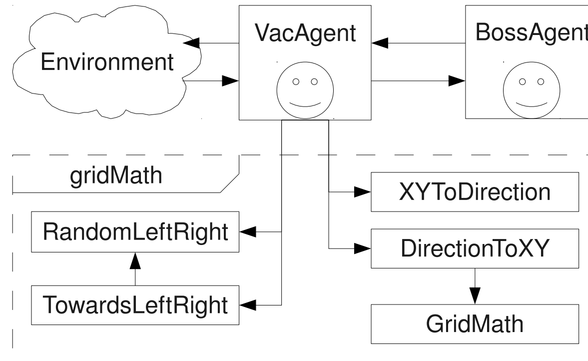

Fig. 2. Outline architecture of an example Jason program, showing AgentSpeak agents with Java internal actions.

4.2 Preliminary evaluation

Of the many possible designs that would solve this problem, ours is just one. We motivate the rest of this paper by asking the question: how maintainable is the example design? Existing object-oriented metrics could of course be applied to the internal actions; but these metrics would not capture any information about the design of the agents themselves, the links between those agents, or the links between agents and their internal actions.

The design could also be evaluated using one of the paradigm-independent techniques discussed in section 3. However, these metrics must be collected manually, which is impractical for large designs and rapid iterative development processes, and likely to result in errors.

To address these shortcomings, we aim to develop a metrics suite for agent programs which operates directly on source code and is amenable to automation.

5 Paradigm-independent metrics

In this section, we present the first steps towards a paradigm-independent suite of metrics, based on a simple theory of software change. To discuss software change precisely, we adopt the terminology of Buckley et al. [8].

5.1 A simple paradigm-independent meta-model

Many programming languages facilitate high-level design, understanding, and re-use by offering the programmer a toolbox of interrelated abstractions. For example, in Java, the available abstractions include packages, classes, interfaces, fields, and methods. During the development process, these abstractions are realized as program text, then processed (for example, by compiling) to form an executable software system.

We define an abstraction as any programming language construct that, when realized as a software artifact:

1. Acts as a container for other software artifacts, such as data, code, and whole or partial realizations of other abstractions.
2. Is named with a textual identifier: a string which has no semantic effect on the program's functionality [1].

In support of this definition, we offer the following definition of containment: if the compile-time deletion of realized abstraction A would also delete part or all of software artifact B, A contains B. To illustrate how these definitions may be applied to real programming languages, figure 5.1 shows the abstractions available in Jason, and their possible containment relationships.

In our current model, artifacts which do not meet the above definition are considered part of their enclosing abstractions. For example, an anonymous Java inner class would be considered part of the named class in which it is defined; and a nameless AgentSpeak plan would be considered part of its enclosing agent. As will be shown in section 5.2, this is a necessary model feature, because an anonymous entity cannot be renamed. Providing fine-grained support for anonymous entities, perhaps by referring to them by source code file name and line number, would add significant complexity to the model and dependency discovery method. Since the principal benefit of this enhancement would be a small gain in the precision of dependency locations, we leave this for future work.

When this model is realized, the resulting containment (or 'aggregation') hierarchy is an acyclic digraph, with the current program as root. We now apply the model to our example Jason program. We use subscripts as follows to denote the type of each abstraction:

- Java: package K, class C, field F, method M
- AgentSpeak: agent A, rule R, belief B, goal G, plan P

A sample of this containment hierarchy is shown in figure 5.1. Note that all the information required for this step was found in, and could easily be programmatically extracted from, the program's source code.

An interesting practical issue was encountered while compiling this containment hierarchy. AgentSpeak agents can give beliefs to other agents, without prior declaration; in our example, two of the vacAgent's possible beliefs ('cleaning' and 'target') are given to it by the bossAgent. This lack of explicit declaration makes it difficult to manually determine the full set of an agent's possible beliefs.

[1] We do not require abstractions to be uniquely named. This allows run-time plan selection in AgentSpeak, and method overloading in Java, to be supported.

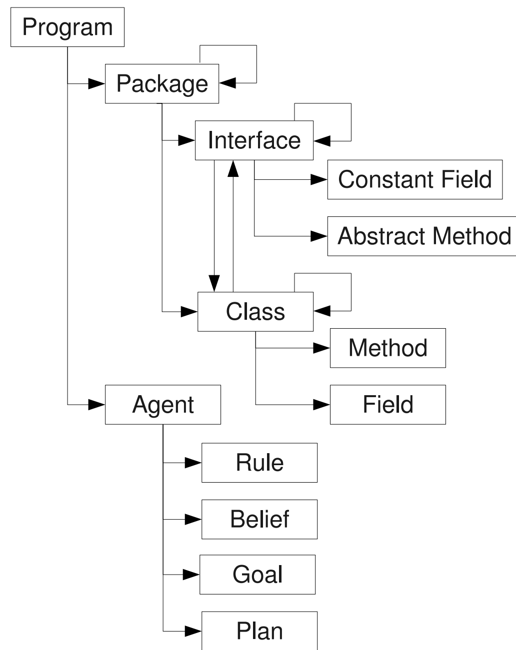

Fig. 3. The abstract containment hierarchy for Jason programs. An arrow from A to B indicates that A *may contain* B.

- vacAgent$_A$
    - blocked$_R$
    - cleaning$_B$
    - target$_B$
    - squareArrived$_P$
    - cleanSquare$_G$
    - cleanHere$_P$
    - cleanNotRequired$_P$
    - ...
- bossAgent$_A$
    - targetArrived$_P$
    - ...
- gridMath$_K$
    - GridMath$_C$
        - angles$_F$
        - toAngle$_M$
        - ...
    - RandomLeftRight$_C$
        - directions$_F$
        - randomLeftRight$_M$
        - execute$_M$
    - TowardsLeftRight$_C$
        - execute$_M$
    - ...

Fig. 4. An extract from a realized containment hierarchy, for an example Jason program.

5.2 Discovering dependencies by refactoring

We now discuss the ripple effects of changes to software, in terms of the above model. Successful software will probably undergo many changes during its lifetime; in devising a suite of structural maintainability metrics, our ideal goal would be to predict, given an existing software design, how much effort and risk will be incurred in making those changes. While acknowledging that changes to software may be radical and unanticipated, in this paper we focus on incremental, evolutionary changes to existing source code, such as the modification of existing program features.

The difficulty of evolving existing code might ideally be estimated by performing example code changes, that are considered representative of the most likely feature modifications. However, for all but the most trivial programs, the list of plausible feature modifications would be vast. Even if program behaviour modifications are excluded, the list of all possible refactorings [13] is considered infinite [26].

Instead of attempting to define a set of representative changes, we propose that useful insight into the 'evolvability' of a code segment could be gained by repeatedly applying a very simple refactoring. Keeping our goal of paradigm-independence in mind, we suggest that just one refactoring is applicable to all conceivable text-based programming languages: the 'rename' refactoring [13, p.273]. This refactoring has the appealing property that it is a semantics-preserving change with no structural effects, which leads to two benefits: for most programming languages it can be easily automated; and the number of distinct modifications required to accommodate a rename refactoring could be said to represent the minimum ripple effect of future modifications.

We propose the following general method to discover the dependencies present within a program:

1. Create a backup copy of the program code
2. For each abstraction realized in a program:
    (a) Rename that abstraction, carefully avoiding new names which are used elsewhere in the program, or have special meaning in the current programming language
    (b) By text replacement only, modify the minimum set of other abstractions, so that the program's original external behaviour is exactly restored
    (c) Note the type and name of the renamed abstraction, and of all other abstractions modified
    (d) Revert all modifications, by restoring from the backup copy

Applying this method reveals a set of dependency mappings between abstractions.

Sample abstraction dependency mappings for our example Jason program are shown in table 5.2. For simplicity, only abstractions directly affected by a modification are shown; a full list of all abstractions affected by a given modification can easily be obtained by propagating upwards through the containment

Abstraction	Set of other abstractions directly affected by rename operation
cleanSquare_G	squareArrived_P, cleanHere_P, $\text{cleanNotRequired}_P$
cleaning_B	cleanHere_P, $\text{cleanNotRequired}_P$, targetArrived_P
RandomLeftRight_C	vacAgent_A, $\text{TowardsLeftRight}_C$
$\text{randomLeftRight()}_M$	$\text{RandomLeftRight}_C.\text{execute}_M$, $\text{TowardsLeftRight}_C.\text{execute}_M$
...	...

Table 1. Partial abstraction dependency mappings for an example Jason program

hierarchy. We use the familiar dot notation to fully-qualify names for clarity where necessary.

Our example results illustrate dependencies

1. within a single agent;
2. within an agent and between two agents;
3. between an internal action and the agent that uses it;
4. within an internal action and between two internal actions.

Of particular interest is the result of the rename operation on the RandomLeftRight class. Two plans within the vacAgent refer to this internal action, and therefore required modification; however, as those plans were not named, they did not appear in our containment hierarchy, and the resulting dependencies were therefore credited to the vacAgent itself. This illustrates a simple benefit of naming AgentSpeak plans: naming allows the location of modification points to be more precisely specified. The TowardsLeftRight class depends on RandomLeftRight by use of the 'extends' Java keyword; thus the object-specific concept of inheritance has been captured in a paradigm-independent way.

5.3 From dependencies to metrics

From dependency mappings such as those described above, a very large number of potentially-useful coupling and cohesion metrics could be devised. However, it is highly desirable that any such metrics should be validated experimentally [18], which is beyond the scope of the current study. Instead, in this section we devise metrics which are generalizations of the well-validated 'Coupling Between Object classes' (CBO) and 'Lack of Cohesion Of Methods' (LCOM) measures [9], returning to the earlier notion of 'coupling between abstractions' [22] and its cohesiveness equivalent.

A central problem in generalizing object-oriented metrics is determining the paradigm-independent equivalents of classes, fields, and methods. Our solution is to adopt the mereological perspective [33], numbering the abstract containment hierarchy of figure 5.1, starting from zero at the most fine-grained level; thus methods, fields, plans, and beliefs all appear at level 0, while classes and agents

both occupy level 1 [2]. This theoretical approach accords with the widely-held view that agents are specializations of objects [30].

Coupling Between Abstractions (CBA) The definition of CBO states that "an object is coupled to another object if one of them acts on the other". We generalize this definition as follows: a level-1 abstraction is coupled to another level-1 abstraction if any dependency exists between those abstractions or their parts. CBA for a level-1 abstraction is thus a count of the number of other level-1 abstractions to which it is coupled.

Lack of Cohesion of Abstractions (LCA) LCOM is defined in terms of the instance variables used by the methods in a class. Since the definition of LCOM has been criticized for its ambiguity regarding dependencies between methods [21], we generalize this definition to include dependencies between all the level-0 abstractions contained within a given level-1 abstraction. LCA for a level-1 abstraction is thus defined as the number of disjoint (i.e. uncoupled) sets of level-0 abstractions it contains, including any dependencies via the enclosing abstraction itself.

To conclude our example, the values of CBA and LCA for all the level-1 abstractions (classes and agents) in our Jason program are shown in table 5.3.

Level-1 abstraction	CBA	LCA
vacAgent	5	1
bossAgent	1	1
GridMath	1	3
RandomLeftRight	2	1
TowardsLeftRight	2	1
DirectionToXY	2	1
XYToDirection	1	1

Table 2. Values of the paradigm-independent metrics *CBA* and *LCA* for level-1 abstractions in an example Jason program.

The full benefit of metrics is difficult to demonstrate using only a small example; metrics are most useful when comparing similar programs, or evaluating parts of a large software design. However, the results for CBA closely match the connections shown in figure 4.1, thus illustrating how important architectural features can be captured quantitatively with this metric. Had no architectural descriptions been available, this information would have been invaluable.

[2] In many situations, this numbering scheme leads to conflicts towards the root of the hierarchy; in our case, the 'program' abstraction could be numbered either 2 or 3. Many languages, including Java, also offer the possibility of nested higher-level abstractions. These issues lead to difficulty in defining specific, theoretically-sound higher-level metrics; in practice, high-level metrics are often calculated simply by taking averages of lower-level values.

The results for LCA are also of interest. GridMap has an LCA value of three, indicating that its three methods have no common dependencies, and it could easily be split into three separate classes. Both agents have the minimum LCA value of one, but this is due to couplings made via anonymous plans, which in our model are indistinguishable from the agent itself. Had all these plans been named, the LCA values for vacAgent and bossAgent would have been three and two respectively. In both cases, the absence of names hid non-obvious design issues, which the LCA metric would otherwise have uncovered.

6 Conclusions

Metrics are increasingly seen as valuable tools which help researchers to compare technologies, and software engineers to understand where future problems may arise. To our knowledge, no metrics for agent-oriented programming languages have yet been proposed. Furthermore, agent programs written in languages such as Jason are often inherently multi-paradigm, with substantial functionality deferred to object-oriented elements. There is thus a clear need for a multi-paradigm metrics suite.

In this paper, we proposed the paradigm-independent coupling and cohesion metrics CBA and LCA, and demonstrated their application to an example Jason program. The theoretical validity of these metrics as predictors of maintainability rests on the hypothesis that the 'rename' refactoring is representative of general software maintenance - which is currently unproven. However, the Chidamber and Kemerer (CK) metrics CBO and LCOM, from which they are generalized, are both well-validated and theoretically grounded. Validation of CBA and LCA would require a large dataset of Jason programs, and access to Jason programming experts, neither of which was available at the time of writing; independent experimental validation is therefore left for future work.

While deriving our proposed metrics, we uncovered a number of issues relating to AgentSpeak programming. The need to discover and locate dependencies in software is not limited to toolmakers; human programmers must do the same when trying to comprehend any unfamiliar code. We therefore make the following suggestions:

1. Plan naming is optional in AgentSpeak. We advocate that plans should be named where possible, arguing that it helps to precisely locate dependencies and other code issues.
2. Explicit declaration of beliefs and goals in AgentSpeak would make it easier to determine an agent's possible mental states. A similar feature is already provided in the Agent Factory AFAPL2 agent programming language by an 'ontology' construct [10].
3. The addition of a higher-level abstraction capability to AgentSpeak, analogous to packages in Java, would solve the mereological difficulty caused by the different depths of the Java and AgentSpeak abstract containment hierarchies, and permit package-specific metrics to be defined and calculated for Jason programs. Though our perspective is one of measurement rather

than agent decomposition and re-use, our results accord with recent work on agent programming language modularity [24] [16].

Our next steps will be to automate the collection of the CBA and LCA metrics, and validate their use experimentally. More generally, the underlying model also needs to be validated, by applying the metrics to other programming languages, both in the agent paradigm and beyond. As noted in section 5.1, our model could also be extended by providing fine-grained support for dependency locations and anonymous artifacts.

Our use of the 'rename' refactoring as a basis for metrics raised some interesting questions. Can other refactorings be defined for agent programming languages? Would their application reveal any further dependencies?

The measures we have derived reflect only a small proportion of the dependency information gathered. Could any other useful predictors of maintainability be derived from it? A large body of literature exists on the relative contributions of different dependency types to coupling and cohesion [15]. A second debate concerns the issue of whether metrics for software maintainability should be applied at a coarse [28] or fine [5] level of granularity. We speculate that the relative locations of dependencies may also be important; intuitively, two multiply-interdependent abstractions will require less future maintenance effort if those dependencies are closely co-located. New multi-paradigm metrics based on our dependency data could shed light on these issues. If links between these metrics and real-world maintainability can be established, this may in turn lead to new recommendations for multi-paradigm programming practices and language design.

Instead of devising new metrics, one interesting alternative would be to provide a flexible means of visualising and summarising the raw dependency data. Such a tool could allow engineers and researchers to focus on the dependency types appropriate to the current context, and aggregate results at an abstraction level of their choosing.

7 Acknowledgements

The authors would like to thank Jim Buckley, Mike Hinchey, and Rebecca Yates for their help in drafting this manuscript.

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre.

References

1. E.B. Allen, S. Gottipati, and R. Govindarajan. Measuring size, complexity, and coupling of hypergraph abstractions of software: an information-theory approach. *Software Quality Journal*, 15(2):179–212, 2007.
2. J. Bach. Good enough quality: Beyond the buzzword. *Computer*, 30(8):96–98, 1997.

3. V.R. Basili, G. Caldiera, and H.D. Rombach. The goal question metric approach. *Encyclopedia of software engineering*, 1:528–532, 1994.
4. F. Bellifemine, G. Caire, and D. Greenwood. *Developing multi-agent systems with JADE*. Springer, 2008.
5. A.B. Binkley and S.R. Schach. Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In *Proceedings of the 20th international conference on software engineering*, pages 452–455. IEEE Computer Society Washington, DC, USA, 1998.
6. R.H. Bordini, L. Braubach, M. Dastani, A.E.F. Seghrouchni, J.J. Gomez-Sanz, J. Leite, G. O'Hare, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems. *Special Issue: Hot Topics in European Agent Research II Guest Editors: Andrea Omicini*, 30:33–44, 2006.
7. S. Bryton and F.B. e Abreu. Towards Paradigm-Independent Software Assessment. *Proc. of QUATIC'2007*, 2007.
8. J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution*, 17(5):309–332, 2005.
9. S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.
10. R.W. Collier and G.M.P. O'Hare. Modelling and Programming with Commitment Rules in Agent Factory. *Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches*, 2009.
11. D.P. Darcy, C.F. Kemerer, S.A. Slaughter, and J.E. Tomayko. The structural complexity of software: an experimental test. *IEEE Transactions on Software Engineering*, 31(11):982–995, 2005.
12. M. Dragone, D. Lillis, R. Collier, and G.M.P. O'Hare. SoSAA: a framework for integrating components & agents. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 722–728. ACM New York, NY, USA, 2009.
13. M Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.
14. A. Garcia, C. Sant Anna, C. Chavez, V.T. da Silva, C.J.P. de Lucena, and A. von Staa. Separation of concerns in multi-agent systems: An empirical study. *Lecture notes in computer science*, pages 49–72, 2004.
15. Brian Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
16. K. Hindriks. Modules as policy-based intentions: modular agent programming in GOAL. *Lecture Notes in Computer Science*, 4908:156–171, 2008.
17. B. Kitchenham and S.L. Pfleeger. Software quality: The elusive target. *IEEE Software*, 13(1):12–21, 1996.
18. B. Kitchenham, S.L. Pfleeger, and N. Fenton. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, 21(12):929–944, 1995.
19. S. Kramer and H. Kaindl. Coupling and cohesion metrics for knowledge-based systems using frames and rules. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 13(3):332–358, 2004.
20. M. Lanza, R. Marinescu, and S. Ducasse. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer-Verlag New York Inc, 2006.
21. W. Li. Another metric suite for object-oriented programming. *The Journal of Systems & Software*, 44(2):155–162, 1998.

22. K. Lieberherr, I. Holland, and A. Riel. Object-oriented programming: an objective sense of style. *ACM SIGPLAN Notices*, 23(11):323–334, 1988.
23. R.E. Lopez-Herrejon and S. Apel. Measuring and Characterizing Crosscutting in Aspect-Based Programs: Basic Metrics and Case Studies. *Lecture Notes in Computer Science*, 4422:423, 2007.
24. N. Madden and B. Logan. Modularity and compositionality in Jason. In *Proceedings of the Seventh International Workshop on Programming Multi-Agent Systems (ProMAS 2009)*, 2009.
25. T. Mens and S. Demeyer. Future trends in software evolution metrics. In *Proceedings of the 4th international workshop on principles of software evolution*, pages 83–86. ACM New York, NY, USA, 2001.
26. T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution*, 17(4):247–276, 2005.
27. MB O'Neal and WR Edwards. Complexity measures for rule-based programs. *IEEE Transactions on Knowledge and Data Engineering*, 6(5):669–680, 1994.
28. J.F. Ramil and M.M. Lehman. Metrics of software evolution as effort predictors - a case study. In *Proc. Int. Conf. Software Maintenance*, pages 163–172, 2000.
29. C. SantAnna, E. Figueiredo, A. Garcia, and C. Lucena. On the Modularity Assessment of Software Architectures: Do my architectural concerns count? In *Proc. International Workshop on Aspects in Architecture Descriptions (AARCH. 07), AOSD*, volume 7. Citeseer, 2007.
30. Y. Shoham. Agent-oriented programming. *Artificial intelligence*, 60(1):51–92, 1993.
31. A. Sipos, N. Pataki, and Z. Porkoláb. On Multiparadigm Software Complexity Metrics. In *MaCS06 6th Joint Conference on Mathematics and Computer Science*, page 85, 2006.
32. R. Subramanyam and M.S. Krishnan. Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, 2003.
33. V.K. Vaishnavi, S. Purao, and J. Liegle. Object-oriented product metrics: A generic framework. *Information Sciences*, 177(2):587–606, 2007.
34. D. Weyns, A. Omicini, and J. Odell. Environment as a first class abstraction in multiagent systems. *Autonomous agents and multi-agent systems*, 14(1):5–30, 2007.
35. M. Wooldridge and N.R. Jennings. Pitfalls of agent-oriented development. In *Proceedings of the second international conference on autonomous agents*, pages 385–391. ACM New York, NY, USA, 1998.
36. J. Zhao, J. Cheng, and K. Ushijima. A Metrics Suite for Concurrent Logic Programs. In *Proc. 2nd Euromicro Working Conference on Software Maintenance and Reengineering*, pages 172–178. Citeseer, 1998.

Towards Reasoning with Partial Goal Satisfaction in Intelligent Agents

M. Birna van Riemsdijk[1] and Neil Yorke-Smith[2,3]

[1] EEMCS, Delft University of Technology, Delft, The Netherlands.
m.b.vanriemsdijk@tudelft.nl
[2] SRI International, Menlo Park, CA, USA.
[3] American University of Beirut, Lebanon. nysmith@aub.edu.lb

Abstract. A model of agency that supposes goals are either achieved fully or not achieved at all can be a poor approximation of scenarios arising from the real world. In real domains of application, goals are achieved over time. At any point, a goal has reached a certain level of satisfaction, from nothing to full (completely achieved). This paper presents a framework for representing partial goal satisfaction in an intelligent agent. The richer representation enables agents to reason about *partial* satisfaction of the goals they are pursuing or that they are considering. In contrast to prior work on partial satisfaction in the agents literature which investigates partiality from a logical perspective, we propose a higher-level framework based on metric functions that represent, among other things, the progress that has been made towards achieving a goal. We present an example to illustrate the kinds of reasoning enabled on the basis of our framework for partial goal satisfaction.

Categories and subject descriptors: I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Intelligent Agents*

General terms: Design; Theory

Keywords: goal reasoning, partial satisfaction, agent programming

1 Introduction and Motivation

This work starts from the observation that existing cognitive agent programming frameworks (e.g., [30, 4, 10]), i.e., programming frameworks in which agents are endowed with high-level mental attitudes such as beliefs and goals, take a 'boolean' perspective on goals: unless achieved completely, the agents have failed to achieve them. Following Zhou et al. [32], we argue that many scenarios would benefit from a more flexible framework in which agents can reason about *partial goal satisfaction*. As others have recognized, it is important that agents can be programmed with this reasoning ability, because often it is not possible for an agent to achieve a goal completely, in the context of all its commitments situated in the resource-bounded real world. A notion of partiality allows to express that

only part of the goal is achieved, and it facilitates, among other possibilities, changing goals such that only a part has to be achieved.

While prior work proposes a logic-based characterization of partiality, in this paper we aim for a general framework for partial goal satisfaction that also allows quantitative notions of partiality. In particular, we propose a framework based on metric functions that represent, among other things, the progress that has been made towards achieving a goal. Agents rescuing civilians from a dangerous area, for example, may have cleared none, some, or all of the area. Progress may be expressed in terms of different kinds of metrics, such as utility, or in terms of a logical characterization. This richer representation enables an agent or group of agents to reason about partial satisfaction of the goals they are pursuing or that they are considering. The more sophisticated behaviour that can result not only reflects the behaviour expected in real scenarios, but can enable a greater total level of goal achievement. For example, an agent might realize that it cannot completely clear a sub-area and inform teammates of the situation; in turn, they adjust their behaviour appropriately, e.g., by coming to assist.

This paper aims to further establish partial goal satisfaction as an important topic of research, and to provide a step towards a metric-based approach that also allows for quantitative notions of partial achievement. This represents a step towards enhancing the capabilities of cognitive agent programming frameworks. Accordingly, we develop an abstract framework for partial goal satisfaction and discuss the concept using an example scenario. We identify *progress appraisal* (the capability of an agent to assess how far along it is in achieving a goal [6]) and *goal adaptation* (the modification of a goal [17, 23, 32]) as the basic types of reasoning that the framework should support. We illustrate how reasoning using partial goal satisfaction can be embedded into a concrete computational framework for the metrics.

The foremost challenge of the work in this paper is conceptual: we identify the main ingredients we believe should be part of an abstract framework for partial goal satisfaction. Through this, we lay foundations for future work, which will address the important technical challenges that have to be faced to concretize the framework and render it suitable for programming a cognitive agent.

2 Background and Related Work

Goal representation. In cognitive agent programming, the concept of a goal has received increasing attention in the past years. Different goal types have been distinguished (see, e.g., [29, 2] for a discussion), including achievement goals and maintenance goals. The former, which have received the most attention in the literature, form the focus of this paper. In the literature, goals have been viewed as declarative and thus as properties of states (i.e., goals-to-be); we take the same perspective [30, 4, 10].

Achievement goals in logic-based cognitive agent programming languages are often represented as a logical formula, expressing a property of the state of the multi-agent system that the agent should try to achieve [30, 31, 4, 29, 10]. The

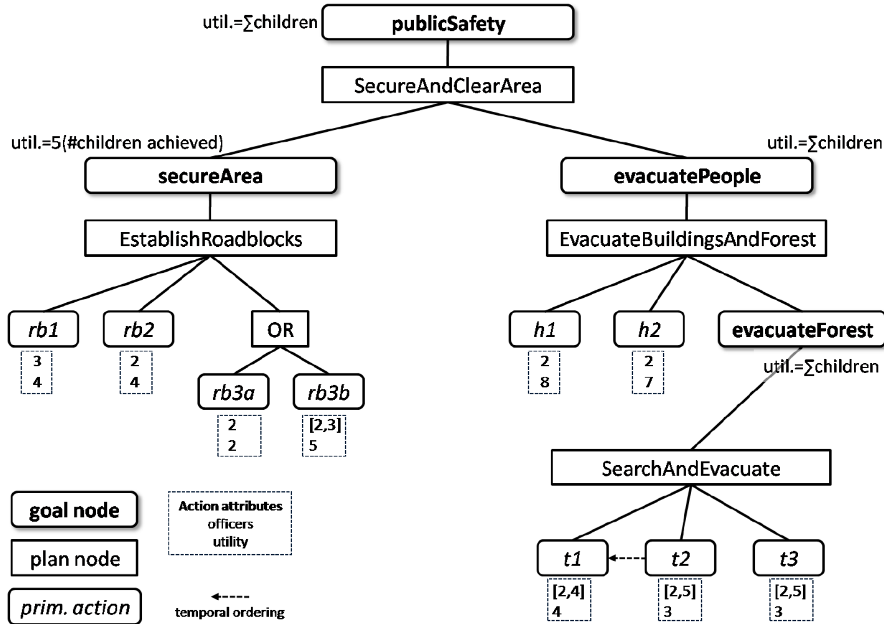

Fig. 1. Goal-plan tree for the scenario.

agent considers the goal to be achieved, if it believes a state has been reached in which the formula is satisfied, according to the semantics of the logic employed. This logic-based approach can induce a binary way of thinking about goals, in which the goal is either achieved or not achieved. While we do not reject that point of view, we suggest in this paper that a framework in which levels of goal satisfaction can be represented enables several useful kinds of reasoning.

Partial achievement. The concept of partial achievement of a goal appears in limited extents in the literature. Whereas goals in agent frameworks and programming languages are not customarily defined to allow for partial satisfaction, philosophically, Holton argues for the existence of "partial intentions" [13], a concept spanning both desires and goals.

In the foundational work of Rao and Georgeff [20] an intention (goal) is dropped if it is achieved, not desired, or now believed by the agent to be impossible. Singh [25] drops an goal if another more important task arises. In these and works that followed them, goal achievement remains a boolean concept.

Haddawy and Hanks made an early study [9], in which a function from propositions to a real number represents the degree of satisfaction of a goal. Indeed, various authors have associated goals with a utility or preference, in the agents literature (e.g., [14, 11], among others) and in the AI planning literature (e.g., [5]), although usually for the purpose of deciding which goals to prioritize or which subset to pursue, or which plan or action to select.

Zhou and Chen adopt instead a logical approach, defining a semantics for partial implication of desirable propositions from a symbolic point of view [31]. Zhou et al. [32] investigate partial goal satisfaction on the basis of this logical semantics, viewing a goal as achieved when a (possibly disjunctive) proposition is achieved according to the logic. They examine in particular application of different notions of partial implication to goal modification in the context of belief change. Although recognizing its value, we do not approach partial satisfaction viewing goals as logical formulas to be achieved. We discuss the relationship between the approaches later.

While van der Hoek et al. [28] explore a related concept, in their logical analysis of BDI intention revision, we aim for more a fine-grained and broader concept. Morley et al. [18] investigate dynamic computation of resource estimates as a partially-complete goal is executed. Again, the representation of a generic concept of partial achievement is not the focus of their work.

Partial plans and goal/plan adaptation. There is a fair amount of work on reasoning with partial *plans*, for instance in plan formation or negotiation (e.g., [17, 8, 15]), as well as in the AI planning literature (e.g., [26]). In the area of multi-agent planning and negotiation, researchers have examined inter-agent communication (e.g., about problems in goal achievement). Kamar et al., for instance, investigate helpful assistance of teammates in pursuit of a plan that could be partially complete [15]. Goal adaptation has received less attention than the concept of goal or plan selection (e.g., [17]), or plan adaptation, the benefits of which are well established [19].

3 Example Scenario

We illustrate by means of an extended example the benefits that a framework for partial goal satisfaction may bring. The scenario is from the domain of crisis management. An accident has occurred in a chemical plant and hazardous chemicals have leaked into the area. The emergency response team must prevent anyone from entering the vicinity of the plant, and evacuate those who are currently in the area. A team of agents will execute a joint plan according to their training. Securing the area is done by setting up road blocks on the three main roads leading to the plant; the third road block can be installed in one of two different places. The two houses within a 3 km radius of the plant must be evacuated. The forest within the range of the chemical leak must be searched and any people brought to safety.

Fig. 1 depicts a *goal-plan tree* (GPT) [27, 3, 18] for the emergency response team in the scenario. A goal-plan tree consists of alternating layers of goal nodes and plan nodes. Goals are depicted in rounded boxes, and plans in square boxes. Goals descending from a plan node are conjunctive: all must be achieved for the plan to be successful. An OR node indicates disjunctive subgoals: achievement of any one renders the plan successful. Thus, the plan *EstablishRoadblocks* is successful when goals *rb1* and *rb2* and at least one of *rb3a* and *rb3b* are achieved.

Primitive actions (leaf goal nodes) are depicted in italicized rounded boxes. The numerical attributes on leaf nodes will be discussed later.

This scenario would benefit from agents being able to reason with partial goal satisfaction. A basic type of reasoning is *progress appraisal* [6]. Progress appraisal is the capability of an agent to assess how far along it is in achieving a goal, i.e., which part of a goal it has already achieved. In the scenario, for example, it may be important for the commander to keep headquarters up-to-date on her progress in setting up the road blocks.

Another, more advanced, type of reasoning with partial goal satisfaction is *goal negotiation*, which has been identified as a key challenge for agents research [16]. Assume, for example, that the team does not have enough members to secure the area and evacuate the forest. The commander may engage in goal negotiation with headquarters, to try to adapt the *publicSafety* goal so only the part that is achievable for the team will have to be pursued. Note that the ability to do *goal adaptation* is thus necessary in order to engage in goal negotiation. The commander suggests to set up only road blocks 2 and 3. However, neglecting road block 1 is not an option according to headquarters, since people may (re-)enter the area, which would lead to a hazardous situation and further evacuation duties. The latter decision is based on an analysis of the importance of achieving the various subgoals. The commander agrees with headquarters that another team will be sent to set up road block 1. Both goal negotiation and adaptation thus require agents to reason about the parts of which a goal is composed.

These kinds of reasoning may occur not only before a goal is adopted, but also during pursuit of a goal. For example, the commander may notice that searching the forest is taking more time than expected, and the team will not be able to search the entire forest before darkness sets in. Rather than abandoning *evacuateForest* entirely because the goal cannot be achieved completely, the team can perform an inferior search of it and achieve it only partially. A decision of whether this is acceptable, or whether it would be better to abandon the forest altogether, depends on an analysis the gains made by achieving the goal only partially—which in this case might be substantial since any person brought to safety is an accomplishment.

This paper provides a high-level framework for partial goal satisfaction that allows a quantitative instantiation, aimed at enabling the kinds of reasoning such as discussed above. After introducing the framework, we mention several other kinds of reasoning that benefit from such a framework for partial satisfaction.

4 Partial Goal Satisfaction and Progress Appraisal

At the heart of conceptualizing partial goal satisfaction is identifying how to define partiality. For this, it is essential to define when a goal is *achieved* (satisfied completely): we cannot define partiality without knowing what complete satisfaction means. In pursuit of our interest in a quantitative framework, moreover, one needs a metric in terms of which (complete) satisfaction is expressed. This metric will be endowed with a partial ordering, to allow an agent to determine

whether a goal is getting closer to completion. We call such a metric the *progress metric* of a goal, and denote it as a set A with partial order $\leq$.[4] A goal then specifies a minimum value $a_{min} \in A$ (called the *completion value*) that should be reached in order to consider the goal to have been completely satisfied. For example, the progress metric for the goal *evacuateForest* might be defined in terms of time, where complete satisfaction is achieved when the forest has been searched for two hours (until it gets dark); or the metric may be defined of a (boolean) proposition such as *isSearched(forest)*; or it may be defined in terms of the number of subgoals achieved (e.g., searching tracks 1, 2, and 3), where complete satisfaction means that all tracks have been searched, etc.

One may consider a wide range of domain-independent metrics, such as TIME, UTILITY, NUMBER OF SUBGOALS, besides domain-dependent metrics such as NUMBER OF ROAD BLOCKS, NUMBER OF PEOPLE BROUGHT TO SAFETY. Besides the metric chosen as the progress metric, the agent (or designer) might have interest in others: e.g., progress may be defined in terms of tracks searched, but time taken could be an additional relevant factor in the team's decisions.

As seen earlier, a fundamental reasoning concerning partial goal satisfaction is *progress appraisal* [6]. An agent should thus be able to determine in a given situation where it is with respect to a progress metric $(A, \leq)$. For example, if TIME is the metric, the agent needs to be able to determine how long it has spent so far. In the case of TIME, the computation from the current state to the time spent is relatively direct. The computation may be more involved for other metrics. In the case of UTILITY, for example, more computation might be needed to determine the current appraised value of utility in terms of other, measurable quantities (i.e., other metrics besides the progress metric). However, in all cases, an agent should be able to determine, given its beliefs about current state, at least an estimation of the value of the progress metric for a goal.

Formally, for a goal with progress metric $(A, \leq)$, we require an agent to have a *progress appraisal function* $\phi : S \to A$, where S is the set of states (i.e., world state and multi-agent system state). In addition, in order to allow determination of whether the completion value $a_{min} \in A$ is reachable given the current state, we normally require the agent to have a *progress upper bound function* $\hat{\phi} : S \times M \to A$ that takes a state $s \in S$, and the *means* $m \in M$ that will be used for pursuing the goal, and yields (an estimation of) the maximum value in A reachable from state s with means m. The upper bound will enable reasoning about the achievability of a goal.

In the abstract framework, we do not further detail the content of the set of possible means M. The content of M will depend on the domain and the concrete agent (programming) framework that is used. Typically, we envisage that M will contain a description of plans and/or resources that can be used to pursue the goal. Sect. 6 contains an example in which we use the goal-plan tree.

[4] Combinations of metrics might be considered, but for simplicity, here we assume quantities are defined in terms of a single metric.

The functions ϕ and $\hat{\phi}$ now allow us to define a *goal template*. The intuition is that each type of goal, such as *secureArea* or *evacuateForest*, has an associated template. On the basis of a goal template, goal instances can be created.

Definition 1 (goal template). *Consider a multi-agent system (MAS) for which the set of possible states is defined as S. Let A be a nonempty set with a partial ordering $\leq$ (the* progress metric*), and let M be a set representing means that can be used for achieving a goal. A* goal template *T is then defined as a tuple $\langle A, M, \phi : S \to A, \hat{\phi} : S \times M \to A\rangle$, where ϕ is the* progress appraisal function*, and $\hat{\phi}$ is the* progress upper bound function.

This notion of goal template may be simplified to consist of only A and ϕ, if $\hat{\phi}$ cannot be provided in a certain case, i.e., when no sensible upper bound can be specified for a goal. Alternatively, it may be extended in various ways. First, the goal template itself may be parameterized to account for variants of the template. For example, depending on the area that has to be secured, the number of road blocks that have to be set up will differ, and this may influence the definition of ϕ and $\hat{\phi}$. Second, one may want to define a goal template for a single goal based on different progress metrics, allowing the agent to choose a progress metric depending on circumstances. We can capture this most simply by having two separate goal templates. Formally relating these templates (for instance by making them siblings in a hierarchy of goal types) is an extension of our basic framework. For reasons of simplicity and space, we leave the pursuit of these extensions for future work.

As noted, in order to simplify definition and computation of ϕ and $\hat{\phi}$, these functions may yield *estimated* values for progress appraisal and the upper bound. In environments that are not fully observable or that are open or dynamic, the agent may not be able to compute precisely the functions. However, an agent must be mindful of the potential adverse effects of estimation. In over-estimation of $\hat{\phi}$ or under-estimation of ϕ, the agent would try to achieve a goal even though it may be impossible to fully satisfy it, or it is already completely satisfied. On the other hand, in under-estimation of $\hat{\phi}$ or over-estimation of ϕ the agent would stop too soon. Thus, while ϕ and $\hat{\phi}$ may yield estimated values, intuitively the agent should estimate the progress upper bound in a state $s \in S$ to be at least the current progress in that state. We call this *coherency* of a goal template, and formally define it as $\forall s \in S, m \in M : \hat{\phi}(s, m) \geq \phi(s)$.

To illustrate Def. 1, consider the goal *secureArea* of the example scenario. In the scenario, the main resource (leaving aside time) is the number of police officers $P = \{0, \ldots, 10\}$. We base the progress metric for the goal on the NUMBER OF SUBGOALS ACHIEVED.

Example 1. The goal template for *secureArea* is: $T_{sa} = \langle \mathbb{R}, P, \phi_{sa}, \hat{\phi}_{sa}\rangle$. Thus, the progress metric is $A = \mathbb{R}$ with its standard $\leq$ ordering. Arbitrarily, we define $\phi_{sa}(s)$ to be 20 if all subgoals have been fully achieved in s (assuming the agent can determine this in each state s), which means that road blocks have been set up and at least one police officer guards each road block, 10 if all road blocks have been set up but not all of them have at least one officer, and 0

otherwise. Let the means include p, the number of officers allocated. We define $\hat{\phi}_{sa}(s,p)$ to be 20 iff the plan *EstablishRoadblocks* can be executed in s and it is executed with at least 6 police officers, i.e., $p \geq 6$, 10 if $1 \leq p < 6$ and the plan can be executed successfully, and 0 otherwise. Computation of the upper bound thus requires determining whether *EstablishRoadblocks* can be executed successfully. This may be done by checking simply the precondition of the plan, or by performing planning or lookahead (compare [3, 12]).

A goal template specifies the progress appraisal and progress upper bound functions. As already addressed above, we need to specify the *completion value* for a goal to specify when it is completely satisfied. In addition, the agent should determine the means that will be allocated for pursuing the goal. The completion value and means together form a goal instance.

Definition 2 (goal instance). *Let* $T = \langle A, M, \phi : S \to A, \hat{\phi} : S \times M \to A \rangle$ *be a goal template. A* goal instance *of* T *is specified as* $(a_{min}, m) : T$*, where* $a_{min} \in A$ *is the completion value, and* $m \in M$ *specifies the means that will be used for achieving the instance.*

Example 2. In the scenario, one goal instance of the goal template T_{sa} for *secureArea* is $g_{sa} = (20, \{0, \ldots, 6\}) : T_{sa}$, expressing that the commander would like to achieve a progress metric value of 20 with no more than six police officers.

Achievement. Using this notion of goal instance, we can easily define when a goal is *achieved* (completely satisfied) in a certain state $s \in S$, namely, when the appraised value of the progress metric in s is at least the completion value.

We will assume that each progress metric $(A, \leq)$ has a (totally ordered) bottom element $\bot_a \in A$ for which $\forall a \in A$ with $a \neq \bot_a$, we have $\bot_a < a$. The bottom element represents a 'zero' achievement level. When a goal instance g is created, it may start partially completed, i.e., $\phi(s) > \bot_A$ where s is the state in which g is created. For example, the road block on road 1 may already be in place, when an instance of *secureArea* is created, because the road was closed for construction.

We can now formally define goal achievement. In addition, we use the progress upper bound function and the means of the goal instance to define a kind of *goal consistency*. In logic-based frameworks for goals [30, 31, 10], an inconsistent goal is not reachable by definition. Achievement in our framework for partial goal satisfaction is similar. We say that a goal instance is *achieved* in a state s if the maximum attainable value of the progress metric from s, given the means of the goal instance, is at least the completion value.

Definition 3 (goal achievement and achievability). *Let* T *be a goal template, let* $(a_{min}, m) : T$ *be an instance of* T*, and let* $s \in S$ *be the current state. The goal instance* $(a_{min}, m) : T$ *is* completely unachieved *iff* $\phi(s) = \bot_A$*,* (completely) achieved *(or* satisfied*) iff* $\phi(s) \geq a_{min}$*, and* partially achieved *otherwise, i.e., iff* $\bot_A < \phi(s) < a_{min}$*. The goal instance is* achievable w.r.t. m *(or simply* achievable*, where the context is clear) iff* $\hat{\phi}(s, m) \geq a_{min}$*.*

For example, the goal instance g_{sa} of Example 2 above is achieved if all road blocks have been set up and each remains guarded by at least one police officer (since in that case the achieved ϕ_{sa} value is 20). It is achievable in any state $s \in S$ since six police officers are allocated for achieving the instance, whence the progress upper bound is 20, equalling the completion value. If less than six officers were allocated, the goal instance would not be achievable since then the agent could maximally attain a ϕ_{sa} value of 10.

4.1 Binary Goal Achievement

We now discuss how our framework relates to logic-based frameworks for (achievement) goals. In the latter, as noted in Sect. 2, the success condition of a goal is usually defined as a logical formula s, which is achieved in a state $s \in S$ if the agent believes s to hold in that state. We show how our definition for partial goal achievement can be instantiated such that it yields the usual binary definition of goal. We abstract from means M.

Definition 4 (binary goal instance). *Let ψ be a logical formula, for which the truth value can be determined in the current MAS state s (denoted as $s \models \psi$ iff ψ is entailed). Let $A = \{\text{false}, \text{true}\}$ with* true > false. *Let $M = \{\epsilon\}$ where ϵ is a dummy element. Let $\phi(s) =$ true if $s \models \psi$ and* false *otherwise, and let $\hat{\phi}(s, \epsilon) =$ true if $\psi \not\models \bot$ and* false *otherwise. Let $T_{bin(\psi)} = \langle A, M, \phi, \hat{\phi} \rangle$. Then we define a* binary goal instance *$\psi = (\text{true}, \epsilon) : T_{bin(\psi)}$.*

Proposition 1 (correspondence). *The instantiation of the partial goal framework as specified in Def. 4, corresponds to the binary definition of goal (Sect. 2) with respect to achievement and consistency.*

Proof. We have to show that achievement and consistency hold in the binary definition of goal, iff achievement and achievability hold in the instantiated partial definition of goal. The goal ψ is achieved in the partial case in some state s iff $\phi(s) \geq a_{min}$, i.e., iff $\phi(s) \geq$ true, i.e., iff $\phi(s) =$ true, i.e., if $s \models \psi$. This is exactly the definition of achievement in the binary case. The goal ψ is achievable in the partial case iff $\hat{\phi}(s, \epsilon) \geq a_{min}$, i.e., iff $\hat{\phi}(s, \epsilon) =$ true, which is precisely the case iff ψ is consistent. □

We envisage an instantiation of our framework with the logic-based characterization of partiality of Zhou et al. [32], where in particular the ordering relation of the progress metric will have to be defined. That is, consider a semantics of partial implication and an alphabet of atoms. Intuitively, we must specify a metric on a set such as propositions over the alphabet, that gives rise to a partial order of the propositions w.r.t. the semantics of implication. Making this instantiation precise will be future research. Indeed, the role of partial implication in connection with subgoals and plans—which we account for in our framework through the computation of metrics in the GPT—has already been noted as a research topic [32].

The instantiation of our framework with a binary goal definition emphasizes that progress metrics need not be numeric. However, if ϕ_g is numeric, the agent

can compute how far it is in achieving a goal as a ratio with the completion value. That is, if T is a goal template with progress appraisal function $\phi : S \rightarrow A$ and $g = (a_{min}, m) : T$ is a goal instance of T, and if quotients in A are defined (e.g., if $A = \mathbb{R}$), then a measure of progress of goal instance g when the agent is in state s is the ratio $\frac{\phi(s)}{a_{min}}$. This metric of % COMPLETE corresponds to the intuitive notion of progress as the percentage of the completion value attained.

5 Goal Adaptation

The previous section outlined an abstract framework for partial goal satisfaction. We have taken progress appraisal as the most basic form of reasoning that such a framework should support. In the motivating scenario, we argued that the framework should support more advanced kinds of reasoning, such as goal negotiation. In this section, we highlight a type of reasoning that we suggest underlies many of these more advanced kinds of reasoning, namely *reasoning about goal adaptation.* Given a goal instance $g = (a_{min}, m) : T$ where T is a goal template, we define *goal adaptation* as modifying a_{min} or m (or both). Note that modifying the plan for g is included in the scope of modifying m.

The reasoning question is how to determine which goals to adapt and how to adapt them. While this is a question that we cannot fully answer here, we analyze the kinds of adaptation and possible reasons for adapting. One important factor that may influence the decision on how, and particularly when, to adapt is the evolution of the agent's beliefs. This aspect is a focus of prior works [22, 32]. Another important factor is the consideration of a *cost/benefit analysis.* We develop our basic framework to support this kind of reasoning.

5.1 Reasons for and Uses of Adaptation

We begin by distinguishing *internal* and *external* reasons for goal adaptation. By internal reasons for we mean those that arise from issues with respect to the goal itself, while external reasons are those that arise from other factors.

More specifically, we see a lack of achievability as a main internal reason for goal adaptation. If a goal instance g is not achievable, it means that its completion value cannot be attained from the current state with the means that are currently allocated. The options without a concept of partial satisfaction are to drop/abort g, to attempt a different plan for g (if possible), to suspend g until it becomes achievable (for example, waiting for more officers to arrive), or to abort or suspend another goal in favour of g. In our framework, the goal instance can be *adapted* to make it achievable by lowering the completion value, which we call *goal weakening*, as well as by the alternative of choosing different means that allows the achievement of the current completion value, e.g., by investing additional resources. Depending on the circumstances, the latter may not always be possible. For example, if the goal is to evacuate people from their houses but it is physically not possible to get to these houses, e.g., because of flooding, it does not matter whether the officers devote more time or personnel.

Several external reasons may lead to goal adaptation. First, a goal instance g may in itself be achievable, but (collective) unachievability of other goal instances may be a reason for adapting g. That is, in practice an agent has only limited resources and it has to choose how it will invest them to achieve a set of current and future goal instances [1, 27]. For example, the agent may decide that another goal instance is more important and needs resources, leading to adaptation of the means of g. In our framework, goal adaptation provides the agent with the option of *partially* suspending, replanning, or abandoning goals. Moreover, progress appraisal helps the agent determine which goals to adapt. For example, it does not seem sensible to drop a goal instance that has a plan that is almost completed and that yields zero utility unless it is completely satisfied.

There are further external reasons for goal adaptation. Second, a particular case is consideration of a new candidate goal instance g': the question of *goal adoption*. Partial satisfaction allows an agent to consider adapting an existing goal instance, or adopting the new instance g' in a weakened form. Third, an agent might be requested by another agent to increase the completion value of a goal instance, which we call *goal strengthening*. For example, the team leader may decide that more time should be spent searching the forest.

Together, progress appraisal and goal adaptation form a basis for higher-level reasoning tasks. We have already discussed goal negotiation (Sect. 3), goal adoption, and avoiding and resolving goal achievement inconsistencies. We now briefly discuss several other kinds of reasoning. First, in order to coordinate their actions, agents should *communicate* about how far they are in achieving certain goals [7, 17, 15]. Progress appraisal provides a principled approach. Second, an agent might realize it cannot achieve a goal completely. Allowing itself to weaken a goal, it can *delegate* part of the goal to other agents. Similarly, delegation may be another option for an agent finding it has achievement difficulties. Related, third, is *reasoning about other agents* and their ability to complete tasks. For example, one agent realizing that another agent is unlikely to fully complete its task(s), irrespective of whether the other agent has acknowledged this.

5.2 Cost/Benefit Analysis

When deciding which goals to adapt and how, we suggest that a cost/benefit analysis can be an important consideration (see also, e.g., [1, 21]). We have already noted the example of an agent pursuing a goal that yields zero utility unless completely satisfied, for which only a small additional amount of effort is required. On the other hand, if an agent has obtained much utility from a goal instance g, compared to that expected when the progress metric of g reaches the completion value, and if much more effort would have to be invested to fully achieve g, it may be sensible to stop pursuit of the goal if resources are needed elsewhere. These kinds of cost/benefit analyses to obtain an optimal division of resources over goals essentially form an optimization problem. While it is beyond the scope of this paper to investigate how optimization techniques can be applied in this context, we do analyze how our framework supports it.

In order to weight up costs and benefits, one needs to know how much it would cost to achieve a certain benefit. The benefit obtained through progress on a goal can be derived in our framework by means of a UTILITY metric $u_g : S \to U$, where U is a set.[4] Note that the progress metric of a goal template might be defined in terms of u_g (as is the case for *publicSafety*), in which case $\phi_g \equiv u_g$ and $A \equiv U$; or u_g might be a different metric (as in the case of *secureArea*).

The incremental benefit obtained when achieving a goal completely is the difference between the benefit at a state s^* for which $\phi(s^*) \geq a_{min}$ (i.e., upon completion of the goal) and the benefit in the current state s_{now}. That is, for a goal instance $(a_{min}, m) : T$, the incremental benefit is $\Delta u = u(s^*) - u(s_{now})$. Note that Δu can only be calculated in this way if differences are defined on U, which will be the case if U is numeric.

The cost associated with obtaining a_{min} can be computed by a COST function $\kappa : S \times M \times S \to C$, where C is a set and $\kappa(s, m, s') = c$ implies that the cost of going from state s with means m to state s' is estimated to be c. Then we can calculate the estimated minimal incremental cost to move from the current state s_{now} to a completion state s^* with means m as $\min_{s' : \phi(s') \geq a_{min}} \kappa(s_{now}, m, s')$.

Supposing that the set that measures benefit, U, and the set that measures cost, C, are mutually comparable—for instance, if both are subsets of $\mathbb{R}$—then the estimates for utility achieved so far, utility expected upon completion, cost so far, and cost to completion can be compared.

6 Towards an Embedding within a Goal Framework

In this section, we illustrate how our metric-based framework for partial goal satisfaction can be applied to a concrete goal representation framework, namely the GPT as introduced earlier. This is a step towards rendering the capabilities within a cognitive agent programming framework. An attraction of the GPT is its representation of goals, subgoals, and plans—which is pertinent for reasoning about the means and the progress in execution of a goal—combined with the annotation of and aggregation of quantities on the tree nodes—which we will use for computation of metrics. Fig. 1 depicted a goal-plan tree for the evacuation scenario. The goal and action nodes correspond to goal instances in our framework; the tree structure gives the plan aspect of their means.

For the reasons just given, we posit that the concept of partially satisfied goals fits naturally into this kind of representation framework for goals. We augment annotations of tree nodes to include metrics about goal (and, where relevant, plan) satisfaction. In the simplest case, this comprises annotating each goal node with values from its progress metric A, as we will explain. The % COMPLETE metric allows normalization of the values.

Progress appraisal. Inference over the tree structure computes and updates metrics by propagation upwards from descendant nodes, in a similar fashion as resource estimates and other information are propagated [3, 24]. For example, the current value of % COMPLETE of a parent plan node may be aggregated

from the values of its child goal nodes. Metrics are aggregated according to their nature and the type of the node. For example, by default, a conjunctive plan node will aggregate % COMPLETE as the arithmetic mean of the children's values, while a disjunctive plan node will aggregate it as the maximum of their values. Mechanisms for aggregation have been explored in the cited literature. Since the algorithms are already parameterizable according to the nature of the quantity (in our case, the metric) and the type of the node, we need not repeat them.

The computation is to be made *dynamically* as the current situation evolves [18, 15]. We assume agents can assess the progress of leaf nodes. For instance, the police officers should believe they know when they have finished clearing a house (and so achieve the utility depicted on each leaf node). Hence, there are two types of metric values attributed onto nodes. The first type are *static*, initial, *a priori* values before execution (as depicted in Fig. 1). These correspond to expected, estimated, or required values, such as the utility expected upon full satisfaction of a goal (i.e., $u(s^*)$), and the resources expected to achieve this. The second type of metric values are *dynamic* estimates computed during execution, such as the utility achieved so far from a goal. For the progress metric of each goal instance g, the static value corresponds to the completion value a_{min} of g, while the dynamic value corresponds to the appraised value $\phi_g(s_{now})$.[5]

6.1 Reasoning in the Example Scenario

The response team commander is given the goal *publicSafety*. The doctrinal plan, *SecureAndClearArea*, involves the two subgoals, *secureArea* and *evacuatePeople*; the two may be achieved concurrently, although the team must be mindful that the public may (re-)enter the incident area until it is secured.

Goal templates, metrics, and goal instances. Recall from Example 1 that the goal template for *secureArea* is $T_{sa} = \langle \mathbb{R}, P, \phi_{sa}, \hat{\phi_{sa}} \rangle$, where the progress metric for T_{sa} is the achievement of its subgoals. The UTILITY metric of T_{sa} can be seen from Fig. 1 to be $u_{sa} = 5 * (\#\text{ achieved subgoals})$. u_{sa} may be of interest as a measure of progress, even though this metric does not *define* the progress (according to police doctrine) nor therefore the completion of the goal.

By contrast to *secureArea*, the progress metric of the initial goal *public-Safety* in the scenario is defined in terms of utility. Its goal template is $T_{ps} = \langle \mathbb{R}, P, u_\Sigma, \hat{u_\Sigma} \rangle$ where $\phi_{ps} \equiv u_\Sigma$ specifies the cumulative utility from the subgoals in the current plan for a goal instance of T_{ps}. This progress metric is computed in the obvious manner by recursively transversing the subtree below the goal instance, summing up the current utility estimates for each goal node. Like-wise, the progress upper bound function, $\hat{u_\Sigma}$, can be computed by a recursive descent through the GPT. An *a priori* estimate can be computed, based on the upper bounds of the static, *a priori* utility attributions on leaf nodes [27, 3,

[5] An agent may be capable of directly computing the value of a metric at a (non-leaf) node. In that case, if the reasoning is consistent and the static values on leaf notes are reliable estimates, then the directly-computed and aggregated values should agree. Where they do not, the agent may resolve the conflict according to which of the two computations it believes is most reliable.

24]. For example, an *a priori* upper bound on *EstablishRoadblocks*, relaxing resource considerations, is $4+4+\max(2,5)=13$. Tighter bounds can be obtained by considering resource limitations and the resulting goal interaction and plan scheduling [27, 24].

Goal adoption. The police commander and her team are tasked with the initial goal *publicSafety*; its goal instance is $(40,10):T_{ps}$. The team of 10, including the commander, has too few officers to meet the expected requirements for the full completion of the three roadblock actions (rb_i) and the two house-clearance actions (h_i), let alone the forest. That is, the goal instance is unachievable (i.e., $\hat{\phi_{ps}} < a_{min}$), as can be seen to be the case by examination of the GPT.

Negotiation, delegation, and requesting help. At first, the commander considers allocating six officers for *secureArea* and weakening the *evacuatePeople* goal by omitting the *evacuateForest* subgoal. This is unacceptable to incident control. After further negotiation, control agrees to send urgently a second team to perform *rb1*. The commander thus allocates four officers for *secureArea*. Hence, the goal instances are $(20,4):T_{sa}$ and $(25,6):T_{ep}$. Two officers will search each house; when done, they will join the forest search.

Appraisal and sharing information. As execution proceeds, updated metric values are computed on the leaf nodes of the GPT and aggregated to parent nodes. This provides a situational assessment for the commander. Searching house 2 is taking longer than anticipated. Should the two officers continue with *h2*, or join those searching the forest? Utility of 4 is estimated achieved from *h2* after 25 minutes have elapsed. The original estimate of UTILITY for completion of the goal was 7; but this was only an *a priori* estimate based on typical experience. The commander appraises that the rate of achieving utility is outweighed by the resources employed, and so calls off the officers from house 2.

This extract from the scenario illustrates the more sophisticated reasoning enabled by and founded on a metric-based notion of partial goal satisfaction that is embedded into a concrete computational framework for the metrics.

7 Conclusion and Next Steps

The contribution of this line of work stems from the recognition of the need for a concept of partial goal satisfaction in cognitive agent frameworks, manifest in terms of the proposal of an abstract framework for partial goal satisfaction that identifies the main necessary ingredients for reasoning based on partial goal satisfaction. Our objective is a representation of partial satisfaction integrated into a reasoning framework, and allowing for a quantitative instantiation, in order that cognitive agent programming frameworks might be enhanced. The benefit of the topic and our approach is more sophisticated reasoning about goals, impacting reasoning about selection, adoption, and pursuit; goal progress appraisal; goal interaction; and inter-agent communication and collaboration.

Although we have indicated how our framework may be concretized in the context of GPTs, more work is needed to flesh out the details and investigate how advanced types of reasoning can be built on top of this basis and integrated into a programming framework. The modifications necessary to the semantics of a language such as GOAL [10] must be established and their correctness proved. To be investigated is how the various functions of our framework can be defined in concrete settings, and how existing work on, e.g., reasoning about resources can be used in this context. Also, while our framework provides the basis for reasoning about goal adaptation, it does not provide algorithms that allow the agent to decide how to adapt, weighing costs and benefits. This is an important area for future research, with just one relevant aspect being how to estimate cost and benefit projection into the future. Lastly, possible extensions are ripe for investigation, such as a logical instantiation with reasoning between goal outcomes, following Zhou et al. [32], inclusion of parameters in goal templates, and relation of templates in a hierarchy.

Acknowledgments

We thank David Martin for discussions and the reviewers for their helpful comments.

References

1. M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 14:349–355, 1988.
2. L. Braubach and A. Pokahr. Representing long-term and interest BDI goals. In *Proc. of ProMAS'09*, 2009.
3. B. J. Clement, E. H. Durfee, and A. C. Barrett. Abstract reasoning for planning and coordination. *JAIR*, 28:453–515, 2007.
4. M. Dastani. 2APL: A practical agent programming language. *JAAMAS*, 16(3):214–248, 2008.
5. M. B. Do, J. Benton, M. van den Briel, and S. Kambhampati. Planning with goal utility dependencies. In *Proc. of IJCAI'07*, 2007.
6. P. J. Feltovich, J. M. Bradshaw, W. J. Clancey, M. Johnson, and L. Bunch. Progress appraisal as a challenging element of coordination in human and machine joint activity. In *Proc. of ESAW'07*, 2007.
7. B. Grosz and S. Kraus. Collaborative plans for complex group action. *Artificial Intelligence*, 86(2):269–357, 1996.
8. B. J. Grosz and L. Hunsberger. The dynamics of intention in collaborative activity. *Cognitive Systems Research*, 7(2–3):259–272, 2006.
9. P. Haddawy and S. Hanks. Representations for decision theoretic planning: Utility functions for deadline goals. In *Proc. of KR'92*, 1992.
10. K. V. Hindriks. Programming rational agents in GOAL. In *Multi-Agent Programming: Languages, Tools and Applications*. Springer, Berlin, 2009.
11. K. V. Hindriks, C. Jonker, and W. Pasman. Exploring heuristic action selection in agent programming. In *Proc. of ProMAS'08*, 2008.

12. K. V. Hindriks, W. van der Hoek, and M. B. van Riemsdijk. Agent programming with temporally extended goals. In *Proc. of AAMAS'09*, 2009.
13. R. Holton. Partial belief, partial intention. *Mind*, 117:27–58, 2008.
14. Z. Huang and J. Bell. Dynamic goal hierarchies. In *Proc. of the 1997 AAAI Spring Symp. on Qualitative Preferences in Deliberation and Practical Reasoning*, 1997.
15. E. Kamar, Y. Gal, and B. J. Grosz. Incorporating helpful behavior into collaborative planning. In *Proc. of AAMAS'09*, 2009.
16. G. Klein, D. D. Woods, J. M. Bradshaw, R. R. Hoffman, and P. J. Feltovich. Ten challenges for making automation a "team player" in joint human-agent activity. *IEEE Intelligent Systems*, 19(6):91–95, 2004.
17. V. Lesser and et al. Evolution of the GPGP/TAEMS Domain-Independent Coordination Framework. *JAAMAS*, 9(1):87–143, 2004.
18. D. Morley, K. L. Myers, and N. Yorke-Smith. Continuous refinement of agent resource estimates. In *Proc. of AAMAS'06*, 2006.
19. B. Nebel and J. Koehler. Plan reuse versus plan generation: A theoretical and empirical analysis. *Artificial Intelligence*, 76(1–2):427–454, 1995.
20. A. S. Rao and M. P. Georgeff. Modeling agents within a BDI-architecture. In *Proc. of KR'91*, 1991.
21. M. Schut, M. Wooldridge, and S. Parsons. The theory and practice of intention reconsideration. *JETAI*, 16(4):261–293, 2004.
22. S. Shapiro and G. Brewka. Dynamic interactions between goals and beliefs. In *Proc. of IJCAI'07*, 2007.
23. S. Shapiro, Y. Lespérance, and H. J. Levesque. Goal change. In *Proc. of IJCAI'05*.
24. P. H. Shaw, B. Farwer, and R. H. Bordini. Theoretical and experimental results on the goal-plan tree problem. In *Proc. of AAMAS'08*, 2008.
25. M. P. Singh. A critical examination of use Cohen-Levesque theory of intentions. In *Proc. of ECAI-92*.
26. D. E. Smith. Choosing objectives in over-subscription planning. In *Proc. of ICAPS'04*, 2004.
27. J. Thangarajah, M. Winikoff, L. Padgham, and K. Fischer. Avoiding resource conflicts in intelligent agents. In *Proc. of ECAI-02*, 2002.
28. W. van der Hoek, W. Jamroga, and M. Wooldridge. Towards a theory of intention revision. *Synthese*, 155(2):265–290, 2007.
29. M. B. van Riemsdijk, M. Dastani, and M. Winikoff. Goals in agent systems: A unifying framework. In *Proc. of AAMAS'08*, 2008.
30. M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative and procedural goals in intelligent agent systems. In *Proc. of KR'02*, 2002.
31. Y. Zhou and X. Chen. Partial implication semantics for desirable propositions. In *Proc. of KR'04*, 2004.
32. Y. Zhou, L. van der Torre, and Y. Zhang. Partial goal satisfaction and goal change: Weak and strong partial implication, logical properties, complexity. In *Proc. of AAMAS'08*, 2008.

Reinforcement Learning as Heuristic for Action-Rule Preferences

Joost Broekens, Koen Hindriks, Pascal Wiggers

Man-Machine Interaction department (MMI)
Delft University of Technology

Abstract. A common action selection mechanism used in agent-oriented programming is to base action selection on a set of rules. Since rules need not be mutually exclusive, agents are often underspecified. This means that the decision-making of such agents leaves room for multiple choices of actions. Underspecification implies there is potential for improvement or optimalization of the agent's behavior. Such optimalization, however, is not always naturally coded using BDI-like agent concepts. In this paper, we propose an approach to exploit this potential for improvement using reinforcement learning. This approach is based on learning rule priorities to solve the rule-selection problem, and we show that using this approach the behavior of an agent is significantly improved. Key here is the use of a state representation that combines the set of rules of the agent with a domain-independent heuristic based on the number of active goals. Our experiments show that this provides a useful generic base for learning while avoiding the state-explosion problem or overfitting.

Categories and subject descriptors: I.2.5 [**Artificial Intelligence**]: Programming Languages and Software; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Intelligent Agents*

General terms: Agent programming languages; Robotics; AI; Methodologies and Languages

Keywords: Agent-oriented programming, rule preferences, reinforcement learning

1 Introduction

Agent platforms, whether agent programming languages or architectures, that are rule-based and use rules to generate the actions that an agent performs introduce the problem of how to select rules that generate the most effective choice of action. Such agent programming languages and architectures are based on concepts such as rules, beliefs, and goals to generate agent behavior. Here, rules specify the agent's behavior. A planning or reasoning engine tries to resolve all rules by matching the conditions and actions with the current mental state of the agent. Multiple instantiations of each rule can therefore be possible. An agent can select any one of these instantiations, resulting in a particular action. So,

the rule-selection problem is analogous to but different from the action-selection problem [13]. Rule selection is about which uninstantiated rule to chose; action selection, in the context of rule-based agent frameworks, is about which instantiated rule to chose. In this paper, when we refer to rule we mean uninstantiated rule, i.e, rules that still contain free variables.

Rule-based agent languages or architectures typically underspecify the behavior of an agent, leaving room for multiple choices of actions. The reason is that multiple rules are applicable in a particular situation and, as a result, multiple actions may be selected by the agent to perform next. In practice, it is often hard to specify rule conditions that are mutually exclusive. Moreover, doing so is undesirable as the BDI concepts used to develop agents often are not the most suitable for optimizing agent behavior. An alternative approach is to optimize agent behavior based on learning techniques.

In this paper we address the following question: how to automatically prioritize rules in such a way that the prioritization reflects the utility of a rule given a certain goal. Our aim is a generic approach to learning such preferences, that can be integrated in rule-based agent languages or architectures. The overall goal is to optimize the agent's behavior given a predefined set of rules by an agent programmer, but our approach can also be used by agent programmers to gain insight into the rule preferences and use these to further specify the agent program. As such, we focus on a useful heuristic for rule preferences. We have chosen reinforcement learning (RL) as heuristic as it can cope with delayed rewards and state dependency. These are important aspects in agent behavior as getting to a goal state typically involves a chain of multiple actions, and rules can have different utility depending on the state of the agent/environment.

We present experimental evidence that reinforcement learning can be used to learn rule priorities that can subsequently be used for rule selection. This heuristic for rule priorities works very well, and results in sometimes optimal agent behavior. We demonstrate this with a set of experiments using the Goal agent programming language [5]. Key in our approach is that the RL mechanism uses a state representation based on a combination of the set of rules of the agent and the number of active goals. Our state representation is as abstract as possible while still being a useful base for learning. We take this approach for two main reasons: (1) we aim for a generic learning mechanism; RL should be a useful addition to all programs, and the programmer should not be bothered by the state representation or state-space explosions; (2) an abstract state helps generalization of the learning result as a concrete state representation runs the risk of over fitting on a particular problem instance.

It is important to immediately explain one aspect of our approach that is different from the usual setup for reinforcement learning. In reinforcement learning it is common to learn the *action* that has to be selected from a set of possible actions. In our approach, however, we will apply reinforcement learning to select an *uninstantiated rule* from a set of rules in an agent program. An uninstantiated rule (called *action rule* in Goal, see also Listing 1) is a generic rule defined by the agent programmer.

```
if goal(tower([X|T])), not(bel(T=[])) then move(X,table)
```

is an example of such a rule. We refer to an instantiated rule as a completely resolved (grounded) version of an action rule generated by the reasoning engine responsible for matching rules to the agent's current state. This also means that the action in an instantiated rule may be selected for execution, as the conditions of the rule have been verified by the engine.

```
if goal(tower([a,b])), not(bel([b]=[])) then move(a,table)
```

is an example of an instantiated rule and the action `move(a,table)` is the corresponding action that may be selected. One instantiated rule thus is the equivalent of one action (as it is completely filled in). Many different instantiated rules may be derived from one and the same program rule, depending on the state. An uninstantiated rule is more generic as it defines many possible actions. We focus on learning preferences for uninstantiated rules.

The paper is organized as follows. Section 2 discusses some related work and discusses how our approach differs from earlier work. In Section 3 we briefly introduce the agent language GOAL and use it to illustrate the rule selection problem. Section 4 presents our approach to this problem based on reinforcement learning and presents an extension of GOAL with a reinforcement learning mechanism. In Section 5 experimental results are presented that show the effectiveness of this mechanism. Finally, Section 6 concludes the paper and discusses future work.

2 Related Work

There is almost no work on incorporating learning mechanisms into agent programming. More generally, BDI agents typically lack learning capabilities to modify their behavior [1], although several related approaches do exist.

With regards to related work, several studies attempt to learn rule sets that produce a policy for solving a problem in a particular domain. Key in these approaches is that the rules themselves are learned, or more specific, rule instantiations are generated and evaluated with respect to a utility function. The best performing rule instantiations are kept and result in a policy for the agent. The evaluation mechanism can be different, for example genetic programming [9] or supervised machine learning [7]. In any case, the main difference is that our approach tries to learn rule preferences, i.e., a priority for pre-existing rules given that multiple rules can be active, while the previously mentioned approaches try to learn rule instantiations that solve a problem.

Other studies attempt to learn rule preferences like we do. However, these approaches are based on learning preferences for instantiated rules [10][4], not preferences for the uninstantiated, generic, rules. Further, the state used for learning is often represented in a much more detailed way [10][4]. Finally, as the state representation strongly depends on the environment, the use of learning mechanisms often involves effort and understanding of the programmer [10].

Reinforcement learning has recently been added to cognitive architectures such as Soar [8] and Act-R [2]. In various respects these cognitive architectures are related to agent programming and architectures. They use similar concepts to generate behavior, using mental constructs such as knowledge, beliefs and goals, are also based on an sense-plan-act cycle, and generate behavior using these mental constructs as input for a reasoning- or planning-based interpreter. Most importantly, cognitive architectures typically are rule-based, and therefore also need to solve the rule (and action) selection problem. For example, Soar-RL has been explicitly used to study action selection in the context of RL [6]. Soar-RL [10] is the approach that comes closest to ours in the sense that it uses a similar reinforcement learning mechanism (Sarsa) to learn rule preferences. As explained above, the key difference is that we attempt to learn uninstantiated rule preferences, while Soar-RL learns preferences for instantiated rules [10]. Another key difference is that we use an abstract rule-activity based state representation complemented with a 'goals left to fulfill' counter, as explained in section 4.2.

Finally, [1] present a learning technique based on decision trees to learn the context conditions of plan rules. The focus of their work is to make agents adaptive in order to avoid failures. Learning a context condition refers to learning when to select a particular plan/action, while learning a rule preference refers to attaching a value to a particular plan/action. Our work is thus complementary in the sense that we do not learn context conditions, but instead propose a learning mechanism that is able to guide the rule selection mechanism itself.

3 The Agent Language GOAL

In this Section we briefly present the agent programming language GOAL and use it to illustrate the rule selection problem in agent languages and architectures. For a more extensive discussion of GOAL we refer the reader to [5]. The approach to the rule selection problem introduced in this paper is not specific to GOAL and may be applied to other similar BDI-based platforms. As our approach involves a domain-independent heuristic based on counting the number of goals that need to be achieved, the language GOAL is however particularly suitable to illustrate the approach as declarative goals are a key concept in the language.

GOAL, for Goal-Oriented Agent Language, is a programming language for programming *rational agents*. GOAL agents derive their choice of action from their beliefs and goals. A GOAL agent program consists of five sections: (1) a knowledge section, called the *knowledge base*, (2) a set of beliefs, collectively called the *belief base*, (3) a set of *declarative* goals, called the *goal base*, (4) a *program section* which consists of a set of *action rules*, and (5) an *action specification section* that consists of a specification of the pre- and postconditions of actions of the agent. Listing 1 presents an example GOAL agent that manipulates blocks on a table.

The knowledge, beliefs and goals of a *GOAL* agent are represented using a knowledge representation language. Together, these make up the mental state of

```
 1  main stackBuilder {
 2   knowledge{
 3     block(a), block(b), block(c).
 4     clear(table).
 5     clear(X) :- block(X), not(on(Y,X)).
 6     tower([X]) :- on(X,table).
 7     tower([X,Y|T]) :- on(X,Y), tower([Y|T]).
 8   }
 9   beliefs{
10     on(a,table), on(b,table), on(c,a), on(d,c).
11   }
12   goals{
13     on(a,d), on(b,c), on(c,table), on(d,b).
14   }
15   program{
16     if goal(tower([X|T])),
17       bel((T=[Y|T1], tower(T)); (T=[], Y=table))
18       then move(X,Y).
19     if goal(tower([X|T])), not(bel(T=[]))
20       then move(X,table).
21   }
22   actionspec{
23     move(X,Y) {
24       pre{ clear(X), clear(Y), on(X,Z) }
25       post{ not(on(X,Z)), on(X,Y) }
26     }
27   }
28  }
```

Table 1. Agent for Solving a Blocks World Problem

an agent. Here, we use *Prolog* to represent mental states. An agent's knowledge represents general conceptual and domain knowledge, and does not change. An example is the definition of the concept `tower` in Listing 1. In contrast, the beliefs of an agent represent the *current state of affairs* in the environment of an agent. By performing actions and possibly by events in the environment, the environment changes, and it is up to the agent to make sure its beliefs stay up to date. Finally, the goals of an agent represent *what* the agent wants the environment to be like. For example, the agent of Listing 1 wants to realise a state where block `a` is on top of block `b`. Goals are to be interpreted as *achievement goals*, that is as a goal the agent wants to achieve at some future moment in time and does not believe to be the case yet. This requirement is implemented by imposing a rationality constraint such that any goal in the goal base must not believed to be the case. Upon achieving the *complete* goal, an agent will drop the goal. The agent in Listing 1 will drop the goal `on(a,b), on(b,c), on(c,table)` if this configuration of blocks has been achieved, and only if the *complete* configuration has been achieved.

As GOAL agents derive their choice of action from their knowledge, beliefs and goals, they need a way to inspect their mental state. GOAL agents do so by means of *mental state conditions*. Mental state conditions are Boolean combinations of so-called basic *mental atoms* of the form **bel**(ϕ) or **goal**(ϕ). For example, **bel**`(tower([c,a]))` is a mental state condition which is true in the initial mental state specified in the agent program of Listing 1.

A GOAL agent uses so-called *action rules* to generate possible actions it may select for execution. This provides for a rule-based action selection mechanism, where rules are of the form **if** ψ **then** $a(\boldsymbol{t})$ with ψ a mental state condition and $a(\boldsymbol{t})$ an action. A mental state condition part of an action rule thus determines the states in which the action $a(\boldsymbol{t})$ may be executed. Action rules are located in the program section of a GOAL agent. The first action rule in this section of our example agent generates so-called *constructive moves*, whereas the second rule generates actions to move a *misplaced* block to the table. Informally, the first rule reads as follows: if the agent wants to construct a tower with `X` on top of a tower that has `Y` on top and the agent believes that the tower with `Y` on top already exists, or believes `Y` should be equal to the table, then it may consider moving `X` on top of `Y`; in this case the move would put the block `Y` *in position*, and it will never have to be moved again. The second rule reads as follows: if the agent finds that a block is misplaced, i.e. believes it to be in a position that does not match the (achievement) goal condition, then it may consider moving the block to the table. These rules code a strategy for solving blocks world problems that can be proven to always achieve a goal configuration. As such, they already specify a *correct* strategy for solving blocks world problems. However, they do not necessarily determine a unique choice of action. For example, the agent in Listing 1 may either move block `d` on top of block `b` using the first action rule, or move the same block to the table using the second action rule. In such a case, a GOAL agent will nondeterministically select either of these actions. It is important for our purposes to note here that the choice of rule is at stake here, and not a particular *instantiation* of a rule. Moreover, as in the blocks world it is a good strategy to prefer making constructive moves rather than other types of moves, the behavior of the agent can be improved by preferring the application of the first rule over the second whenever both are applicable. It is exactly this type of preference that we aim to learn automatically.

Finally, to complete our discussion of GOAL agents, actions are specified in the action specification section of such an agent using a STRIPS-like specification. When the preconditions of the action are true, the action is executed and the agent updates its beliefs (and subsequently its goals) based on the postcondition. Details can be found in [5].

As illustrated by our simple example agent for the blocks world, rule-based agent programs or architectures may leave room for applying multiple rules, and, as a consequence, for selecting multiple actions for execution. Rule-based agents thus typically are *underspecified*. Such underspecification is perfectly fine, as long as the agent achieves its goals, but may also indicate there is room for improvement of the agent's behavior (though not necessarily so). The problem of optimizing the behavior of a rule-based agent thus can be summarized as follows, and consists of two components: First, solving a particular task efficiently depends on using the appropriate rule to produce actions (the rule selection problem) and, second, to select one of these actions for execution (the action selection problem). The latter problem is actually identical to selecting an *instantiated* rule where all variables have been grounded, as instantiated rules

that are applicable yield unique actions that may be executed. Uninstantiated rules only yield *action templates* that need to be instantiated before they can be executed.

In this paper we explore a generic and fully automated approach to this optimization problem based on learning, and we propose to use reinforcement learning. Although reinforcement learning is typically applied to solve the action selection problem, here instead we propose to use this learning technique to (partially) solve the rule selection problem. The reason is that we want to incorporate a *generic* learning technique into a rule-based agent that does not require domain-specific knowledge to be inserted by a programmer. As we will show below, applying learning to the rule selection problem in combination with a domain-independent heuristic based on the number of goals still to be achieved provides just such a mechanism.

4 Learning to Solve the Rule Selection Problem

In this Section, we first briefly review some of the basic concepts of reinforcement learning, and then introduce our approach to the rule selection problem based on learning and discuss how we apply reinforcement learning to this problem. We use the agent language Goal to illustrate and motivate our choices.

4.1 Reinforcement Learning

Reinforcement Learning is a mechanism that enables machines to learn solutions to problems based on experience. The main idea is that by specifying *what* to learn, RL will figure out *how* to do it. An approach based on reinforcement learning assumes there is an environment with a set of possible states, S, a reward function $R(S)$ that defines the reward the agent receives for each state in the environment, and a set of actions A that enable to effect changes to the environment (or an agent in that environment) and move the environment from one state to another according to the state transition function $T(S, A) \rightarrow S$. An RL mechanism then learns a value function, $V(S, A)$, that maps actions in states to values of those actions in that state. It does so by propagating back the reinforcement (reward) received in later states to earlier states and actions, called *value propagation*. RL should do this in such a way that the result of always picking the action with the highest value will lead to the best solution to the problem (the best sequence of actions to solve the problem is the sequence with the highest cumulative reward). Therefore, RL is especially suited for problems in which the solution follows only after a sequence of actions and in which the information available for learning takes the form of a reward (e.g. pass/fail or some utility value).

In order for RL to learn a good value function, it must explore the state space sufficiently, by more or less randomly selecting actions. Exploration is needed to gather a representative sample of interactions so that the transition function T (in case the model of the world is not known) and the reward function R can be

learned. Based on T and R, the value function V is calculated. After sufficient exploration, the learning agent switches to an exploitation scheme. Now the value function is used to select the action with highest predicted cumulative reward (the action with the highest $V(s, a)$). For more information on RL see [12].

4.2 GOAL-RL

The idea is to use RL to learn values for the rules in an agent program or architecture, so that a priority of rules can be given at any point during the execution of the agent. Here, we use GOAL to illustrate and implement these ideas, and we call this RL-enabled version GOAL-RL. The basic idea of our contribution is that the GOAL interpreter determines which rules are applicable in a state, while RL learns what the values for applying these same rules are in that state. GOAL will then again be responsible for using these values in the context of rule selection. Various selection mechanisms may be used, e.g., selecting the best rule greedy, or selecting a rule based on a Boltzmann distribution, etc. This setup combines the strenghts of a qualitative, logic-based agent platform such as GOAL with the strengths of a learning mechanism such as reinforcement learning.

RL needs a state representation for learning. Unfortunately, using the agent's mental state or the world state, as is typically done in RL, quickly leads to intractably large state spaces and makes the solutions (if they can be learned at all) domain and even problem-instance specific. Still, our goal is to create a domain-independent mechanism that takes the burden of finding a good rule selection mechanism away from the programmer.

We propose the following approach. Instead of starting to train with a state representation that is as close as possible to the actual agent state, and make that representation more abstract in case of state explosion problems, as is common in RL, we start with a representation that is very abstract, while still being meaningful and specific enough to be useful for learning rule preferences. The benefits of this choice are twofold. First, a trained RL model based on such an abstract state and action representations is potentially more suitable for reuse in different domains and problem instances (learning transfer). Second, by using an abstract state our approach is less vulnerable to large state-spaces and the state-space explosion problem, and, consequently, will learn faster.

The state representation we propose is composed of the following two elements. First, our state representation contains the set of rule-activation pairs itself (i.e. the list of rules and whether a rule is applicable or not). However, for many environments this representation does not contain enough information for the RL algorithm to learn a good value function. Essentially, what is missing is information that guides the RL algorithm towards the end goal. A state representation that only keeps track of the set of rules that are and are not applicable does not contain any information about the approriateness of rules in a particular situation. We add such information by including a second element in the state representation: a version of a well-known progress heuristic used also in planning. The heuristic, which is easily implemented in an agent language or architecture that keeps track of the goals of an agent explicitly, is to count the

number of subgoals that still need to be achieved. This is a particularly easy way to compute the so-called *sum cost heuristic* introduced in [3]. Due to its simplicity this heuristic causes almost no overhead in the learning algorithm. This heuristic information is added to the state used by the reinforcement learning mechanism in order to guide the learning. Adding a heuristic like this will keep the mechanism domain independent, but gives useful information to the RL mechanism to differentiate between states.

Even with this heuristic many states differentiated by the agent itself are conflated in the limited number of states used by the reinforcement learner. Such a state space reduction will sometimes prevent the algorithm from finding optimal solutions (as many RL mechanisms, including the one we use, assume a Markovian state transition). It should be noted, though, that we are not aiming for a perfect learning approach that is always able to find optimal solutions. Instead, we aim for an approach that provides two benefits: it is generic and therefore poses no burden on the programmer, and the approach is able to provide a significant improvement of the agent's behavior, even though this may still not be optimal (optimal being the smallest number of steps possible to solve a problem). The approach to learn rule preferences thus should result in significantly better behavior than that generated by agents that do not learn rule preferences. In the remainder of this paper, we will study and demonstrate how well the domain-independent approach is able to improve the behavior of agents acting in different domains.

In more detail, the approach introduced here consists of the following elements. A state s is a combination of the number of subgoals the agent still has to achieve and the set of rule states. A rule state is either 0, 1 or 2, where 0 means the rule is not active, 1 means there is an instantiation of the rule in which the rule's preconditions are true and 2 means there is an instantiation in which also the preconditions for the action the rule proposes are true meaning the rule fires. For example, if a program has a list of 3 rules, of which the last two in the program fire while the agent still has 4 subgoals to achieve, the state equals to $s = 022 : 4$. An action is represented by a hash based on the rule (in our case simply the index of the rule in the program list; so the action uses the same hash as the rule in the rule-activation pairs used for the state). For example, if the agent would execute an action coming from the first rule in the list, the action equals to $a = 0$, indicating that the agent has picked the first rule for action generation. In our setup, the reward function R is simple. It defines a reward $r = 1$ when all goals are met (the goal list is empty) and $r = 0$ otherwise. The current and next state-action pairs (s, a) and (s', a') are used together with the received reward r' as input for the value function V. A transition function T is learned based on the observed state-action pairs (s, a) and (s', a'). The transition function is used to update the value function according to standard RL assumptions, with one exception: the value for a state-action pair (s, a) is updated according to the probabilistically correct estimate of the occurance of (s', a'), not the maximum. In order to construct the probabilities, the agent counts state occurrences, $N(s)$, and uses this count in a standard weighting mechanism. Values of states are

updated as follows:

$$RL(s,a) \leftarrow RL(s,a) + \alpha \cdot (r - RL(s,a)) \quad (1)$$

$$V(s,a) \leftarrow RL(s,a) + \gamma \cdot \sum_i V(s_{a_i}, a_i) \frac{N(s_{a_i}, a_i)}{\sum_j N(s_{a_j}, a_j)} \quad (2)$$

So, a state-action pair (s,a) has a learned reward $RL(s,a)$ and a value $V(s,a)$ that incorporates predicted future reward. $RL(s,a)$ converges to the reward function $R(s,a)$ with a speed proportional to the learning rate α (set to 1 in our experiments). $V(s,a)$ is updated based on $RL(s,a)$ and the weighted average over the values of the next state-action pairs reachable by action $a_{1...i}$ (with a discount factor of γ, set to 0.9 in our experiments). So, we use a standard model-based RL approach [12], with an update function comparable to Sarsa [11].

5 Experiments

In order to assess if rule preferences can be learned using RL with a state representation as described, we have conducted a series of experiments. The goal of these experiments was to find out the sensitivity of our mechanism with respect to (a) the problem domain (we tested two different domains), (b) different problem instantiations within a domain (e.g. random problems), (c) rules used in the agent program (different rule sets fire differently and thus result in both a different state representation and different agent behavior), (d) different goals (a different goal implies a different reward function because $R(s) = 1$ only when all goals are met).

In total we tested 8 different setups. Five setups are in an environment called the *blocks world*, in which the agent has to construct a goal state consisting of a predefined set of stacks of numbered blocks from a start state following standard physics rules (block cannot be removed from underneath other blocks). The agent can grab a block from and drop a block at a particular stack. In principle, it can build infinitely many stacks (the table has no bounds). The agent program lists two rules.

Three setups were in the *logistics domain* in which the agent has to deliver two orders each consisting of two different packages to two clients at different locations. In total there are three locations, with all packages at the starting location and each client at a different location. A location can be reached directly in one action. The agent can load and unload a package as well as goto a different location. The agent program lists five rules.

5.1 Setup

Each experiment consisted of a classic learning experiment in which a training phase of 250 trials (random rule selection) was followed by a exploitation phase of 30 trials (greedy rule selection based on learned values). For each experiment we

present a bar graph showing the average number of actions needed to solve the problem during the training phase (reflecting the goal agent as it would perform on average without learning ability) and during the exploitation phase (reflecting the solution including the trained rule preferences). As a measure of optimality we also show the minimum number of actions needed in one trial to get to a goal state as observed during the first 250 trials (which equals the minimum number of steps to reach a solution, except in Figure 3 as explained later). This number is shown in the bar graphs as reference number for the optimality of the learned solution.

5.2 Blocks world experiments

Five experiments were done using the blocks world. As described in section 3, there are two rules for the $GOAL$ agent, one rule designed to correctly stack blocks on goal stacks (constructive rule) and the other designed to put ill-placed blocks on the table (deconstructive move). Given these two rules, it is easy to see (and prove) that given a choice between the constructive and deconstructive move, the constructive move is always as good as the deconstructive one. It involves putting blocks at their correct position. These blocks do not need to be touched anymore. A deconstructive move involves freeing underlying blocks. This might be necessary to solve the problem, but the removed blocks might also need to be moved again from the table to their correct place at a goal stack.

The first experiment is a test, constructed to find out if the $GOAL$-RL agent can learn the correct rule preferences for a fundamental three-blocks problem. In this problem, three blocks need to be put on one stack starting with C, BA ending with the goal stack ABC. The agent should learn a preference for the constructive move, as this allows a solution of the problem in two moves ($B > C$ and $A > BC$), while the deconstructive move needs three ($A > Table$ then $B > C$ and $A > BC$). Indeed, the agent learns this preference, as shown in Figure 1.

The reward in the last experiment comes rather quickly, and the state transitions are provably Markovian, so the positive learning result presented here is not surprising. In the second experiment, we tested if a reward given at a later stage together with a more complex state-space would also give similar results. We constructed a problem of which it is clear that constructive moves are better than deconstructive moves: the inverse-tower problem. Here, the agent is to inverse a tower $IHGFEDCBA$ to $ABCDEFGHI$. Obviously, constructive moves are to be preferred as they build a correct tower, while deconstructive moves only delay building the tower. The rules used by the agent are the same as in the previous experiment. As can be seen in Figure 1, the $GOAL$-RL agent is able to learn the correct rule preferences and thereby produce the optimal solution.

As one of the reasons for choosing an abstract state representation is to find out if this helps learning a solution to multiple problem instances with a problem domain, not just the one trained for, we set up a third experiment based on tower building problem in which the starting configuration is random. This means

that at each trial the agent is confronted with a different starting configuration but always has $ABCDEFGHI$ as goal stack. Being able to learn the correct preferences for the rules in this case involves coping with a large amount of environment states that are mapped to a much smaller amount of rule-based states. We have kept the goal static to be able to interpret the result. If the goal is to build a high tower, constructive moves should be clearly preferred over deconstructive ones. Therefore we know that in this experiment the constructive move is clearly favorite and the learning mechanism should be able to learn this. As shown in Figure 2 the agent can indeed learn to generalize over the training samples and learn rule preferences. Note that if we would have taken a state representation more directly based on the actual world (e.g., the current blocks configuration), this generalization is difficult as each new configuration is a new state, and in RL unseen states cannot be used to predict values (unless a RL mechanism is used that uses some form of state feature extraction). Therefore, this result that shows that our approach is able to optimize rule selection in a generic way.

Up until now, the two rules of the agent are relatively smart. Each rule helps solving the problem, i.e., each rule moves forward towards the goal, as even the deconstructive rule never removes a block from a goal stack. In the next experiment we changed the deconstructive move to one that always enables the agent to remove a block form any stack. This results in a *dumb* tower building agent as it can deconstruct correct towers. For this agent to learn correct preferences, it needs to cope with much longer action sequences before the goal is reached as well as many cycles in the state transitions (e.g., when the agent undoes a constructive move). As shown in Figure 2, left and middle, the agent can learn the correct rule preferences and converge to the optimal solution. This is an important result as it shows that the mechanism can cope with different rule sets solving the same problem, as well as optimize agent behavior given a rule set that is clearly sub-optimal (the *dumb* deconstructive move).

In our last experiment with the blocks world, we evaluated whether the learning mechanism is sensitive to the goal itself. It is based on the inverse tower problem, with one variation: instead of having one high tower as goal, we now have three short towers ABC, DEF, GHI as goal stacks as well as a random starting configuration. This variation thus de-emphasizes the merit of constructive moves for the following reason. In order to solve the problem from any random starting configuration, the agent also has to cope with those situations in which one or two long towers are present at the start. These towers need to be deconstructed. As such, even though constructive moves are never worse than deconstructive moves, deconstructive moves become relatively more valuable. As shown in Figure 2, right, the agent still improves the agent behavior significantly, but is not able to always learn the optimal solution. As such our learning approach provides a useful heuristic for rule preferences. The decrease in learning effectiveness is due to the abstractness of the state representation. In the previous experiments, the agent's RL mechanisms could know where it was building the tower, as the number of active subgoals (incorrectly placed blocks)

decreases with each well-placed block. In this experiment, however, the number of active subgoals does not map to the environment state in the same fashion (all three towers contribute to this number, but it is impossible to deduce the environment state based on the number of subgoals: e.g., the number 6 does not reflect that tower one and two are build and we are busy with tower three). This means that there is more *state-overloading* in the last experiment, more risk at non Markovian state transitions, hence the RL mechanism will perform worse.

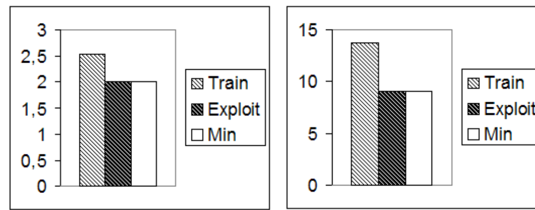

Fig. 1. Left: three-block test. Right: inverse tower.

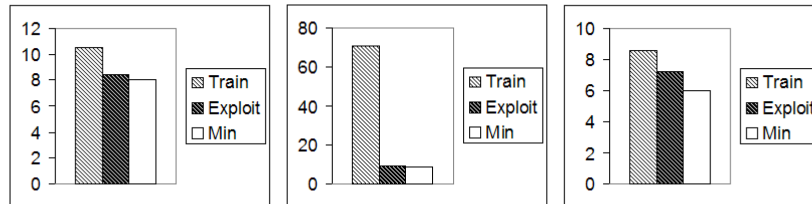

Fig. 2. Left: random start tower. Middle: random start tower dumb. Right random start 3 towers.

5.3 Logistics domain experiments

In this set of three experiments we test the behavior of our mechanism in a different domain. The domain is called the *logistics domain.* As explained above, this domain involves a truck that needs to distribute from a central location two different orders containing two different items to two clients, making it a total of four items to be delivered. A truck can move between three locations (client 1, client 2 and the distribution center). The agent has two goals: deliver order 1, and deliver order 2. It can pick up and drop an item. When two items are delivered, a subgoal is reached. The agent has five rules, two of which handle pickup, one handles dropping, 1 handles moving to a client, and one handles moving to the distribution center.

In the first experiment, we tested if the agent can learn useful preferences in this domain. As Figure 3, left and middle, shows, it can. This suggests that our results are not specific to a single domain.

In the second experiment, we modified the rule that controls moving to clients such that it also allows the truck to move to clients when empty (the *dumb* delivery truck). This mirrors the dumb tower builder in the blocksworld as it significantly increases the average path to the goal state and it introduces much more variation in the observed states (more random moves). As shown in Figure 3, left and middle, the agent can also learn rule preferences that enable it to converge to the optimal solution. We would like to note that the average learning result is better than the minimum result observed during exploration. This shows that the learned rule preferences perform a strategy that is better than any solution tried in the 250 exploration trials. In other words, learning based on rule-based representations can generalize to a better solution than observed during training.

In the last experiment we manipulated a last important factor: the reward function $R(s)$. In the previous two experiments, the agent was positively reinforced when the last item had been delivered. In this experiment, the agent is reinforced when it returns to the distribution center after having delivered the last item. As shown in Figure 3, right, this results in a suboptimal strategy, although still far better a strategy than the standard *GOAL* agent. This shows that the mechanism is influenced by the moment the reward is given, even if from a logical point of view this should not matter. The reason for this is simple (and resembles the one proposed for the slightly worse performance in the last blocksworld experiment). Due to our abstract state representation, the RL mechanism of the agent cannot differentiate between a state in which it just returned to the distribution center after delivering the *last* item of the last order versus the *first* item of the last order. This means that both environment states are mapped to the same RL state. This RL state receives a reward, and therefore returning to the distribution center gets rewarded. As such, the agent emphasizes returning to the distribution center and learns the suboptimal solution in which it picks up an item and brings it to the client as soon as possible in order to get to the center ASAP because that is where the reward is. The best strategy is of course to pick both items for a client and then move to the client. However, as the RL mechanism cannot differentiate between two important states, it cannot learn this solution. This clearly shows a drawback of a too abstract state representation. However, the drawback is relative, as the agent still performs much better than the standard *GOAL* agent, showing that even in this case our mechanism is useful as a rule preference heuristic.

6 Conclusion

In this paper we have focused on the question of how to automatically prioritize rules in an agent program. We have proposed an approach to exploit the potential for improvement in rule-selection using reinforcement learning. This approach

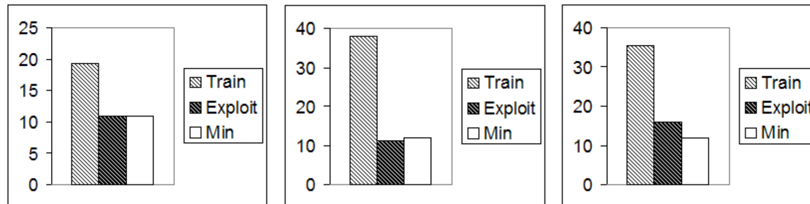

Fig. 3. Left: delivery world. Middle: delivery world dumb. Right: delivery world manipulated $R(s)$

is based on learning state-dependent rule priorities to solve the rule-selection problem, and we have shown that using this approach the behavior of an agent is significantly improved. We demonstrate this with a set of experiments using the *GOAL* agent programming language, extended with a reinforcement learning mechanism. Key in our approach, called *GOAL*-RL, is that the RL mechanism uses a state representation based on a combination of the set of rules of the agent and the number of active goals. This state representation, though very abstract, still provides a useful base for learning. Moreover, this approach has two important benefits: (1) it provides for a generic learning mechanism; RL should be a useful addition to all programs, and the programmer should not be bothered by the state representation or state-space explosions; (2) an abstract state helps generalizing the learning result as a concrete state representation runs the risk of over fitting on a particular problem instance. One of the advantages is that it does not involve the agent programmer or the need to think about state representations, models, rewards and learning mechanisms. In the cases explored in our experiments the approach often finds rule preferences that result in optimal problem solving behavior. In some case the resulting behavior is not optimal, but is still significantly better than the non-learning agent.

Given that we have implemented a very generic, heuristic approach there is still room for further improvement. Two topics are particularly interesting for future research. First, we want to investigate whether adding other domain-independent features and making the state space in this sense more specific may improve the learning even more. Second, we want to investigate whether the use of different learning mechanisms that are better able to cope with non Markovian worlds and state overloading such as methods based on a partially observable Markov assumption (POMDP) will improve the performance.

7 Acknowledgments

This research is supported by the Dutch Technology Foundation STW, applied science division of NWO and the Technology Program of the Ministry of Economic Affairs. It is part of the Pocket Negotiator project with grant number VIVI-project 08075.

References

1. S. Airiau, L. Padham, S. Sardina, and S. Sen. Enhancing adaptation in bdi agents using learning techniques. *International Journal of Agent Technologies and Systems*, 1(2):1–18, 2009.
2. J. R. Anderson and C. Lebiere. *The atomic components of thought*. Lawrence Erlbaum, Mahwah, NY, 1998.
3. B. Bonet, G. Loerincs, and H. Geffner. A robust and fast action selection mechanism for planning. In *Proceedings of AAAI-97*, pages 714–719, 1997.
4. S. Deroski, L. De Raedt, and K. Driessens. Relational reinforcement learning. *Machine Learning*, 43(1):7–52, 2001.
5. K. V. Hindriks. Programming Rational Agents in GOAL. In *Multi-Agent Programming: Languages, Tools and Applications*, chapter 4, pages 119–157. Springer, 2009.
6. E. Hogewoning, J. Broekens, J. Eggermont, and E. Bovenkamp. Strategies for Affect-Controlled Action-Selection in Soar-RL. In *Nature Inspired Problem-Solving Methods in Knowledge Engineering*, pages 501–510. 2007.
7. R. Khardon. Learning action strategies for planning domains. *Artificial Intelligence*, 113:125–148, 1999.
8. J. E. Laird, A. Newell, and P. S. Rosenbloom. Soar: an architecture for general intelligence. *Artif. Intell.*, 33(1):1–64, 1987.
9. J. Levine and D. Humphreys. Learning action strategies for planning domains using genetic programming. In *Applications of Evolutionary Computing*, pages 50–55. 2003.
10. S. Nason and J. E. Laird. Soar-rl: integrating reinforcement learning with soar. *Cognitive Systems Research*, 6(1):51–59, 2005.
11. G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical report, Cambridge University Engineering Department., 1994.
12. R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, 1998.
13. T. Tyrrell. *Computational Mechanisms for Action Selection*. Phd, University of Edinburgh, 1993.

Author Index

Artikis, Alexander 5

Behrens, Tristan 37
Bordini, Rafael 37
Braubach, Lars 37
Broekens, Joost 85

Carr, Hugo 5
Collier, Rem 53

Dastani, Mehdi 37
Dix, Jürgen 37

Hübner, Jomi 37
Hindriks, Koen 37, 85

Jordan, Howell 53

Kraus, Sarit 3

O'Hare, Gregory 1

Pitt, Jeremy 5
Piunti, Michele 21
Pokahr, Alexander 37

Ricci, Alessandro 21

Santi, Andrea 21

van Riemsdijk, M. Birna 69

Wiggers, Pascal 85

Yorke-Smith, Neil 69