Matteo Baldoni, Jörg P. Müller
Ingrid Nunes, Rym Zalila-Wenkstern (eds.)

Engineering Multi-Agent Systems

Fourth International Workshop, EMAS 2016
Singapore, May 9th and 10th, 2016

Workshop Notes

EMAS 2016 Home Page:
`http://www.utdmavs.org/emas2016/`

Preface

The engineering of multi-agent systems (MAS) is a multi-faceted, complex task. These systems consist of multiple, autonomous, and heterogeneous agents, and their global behavior emerges from the cooperation and interactions among the agents. MAS have been widely studied and implemented in academia, but their full adoption in industry is still hampered by the unavailability of comprehensive solutions for conceiving, engineering, and implementing these systems.

Although much progress has been made in the development of multi-agent systems, the systematic engineering of large-scale MAS still poses many challenges. Even though various models, techniques and methodologies have been proposed in the literature, researchers and developers are still faced with the common questions:

- Which architectures are suitable for MAS?
- How do we specify, design, implement, validate and verify, and evolve our systems?
- Which notations, models and programming languages are appropriate?
- Which development tools and frameworks are available?
- Which processes and methodologies can integrate all of the above and provide a disciplined approach to the rapid development of high-quality MAS?

Existing approaches address the use of common software engineering solutions for the conception of MAS, the use of MAS for improving common software engineering tasks, and also the blending of the two disciplines to conceive MAS-centric development processes.

The International Workshop on Engineering Multi-Agent Systems (EMAS) provides a comprehensive venue, where software engineering, MAS and artificial intelligence researchers can meet together, discuss different viewpoints and findings, and share them with industry. EMAS was created in 2013 as a merger of three separate workshops (with overlapping communities) that focused on the software engineering aspects (AOSE), the programming aspects (ProMAS), and the application of declarative techniques to design, program, and verify MAS (DALT). The workshop is traditionally co-located with AAMAS (International Conference on Autonomous Agents and Multiagent Systems) which in 2016 takes places in Singapore.

This year's EMAS workshop is held as a one-and-a-half day event. 14 papers were submitted to the workshop and each submission received, received at least three reviews, the program committee selected 13 papers for presentation. The program also includes two invited talks. T he first "Implementing Norms Why is it so difficult?", by prof. Frank Dignum from the Universiteit Utrecht, is held jointly with the COIN workshop. The second, by prof. Jaime Simão Sichman, is titled "Designing and Programming Multiagent Organizations." We are confident that the talks offer an interesting perspective of the work that has been done for conceiving sound and complex MAS, and they will also offer the opportunity for fruitful and interesting discussions.

We would like to thank the members of the Program Committee for their excellent work during the reviewing phase. We also acknowledge the EasyChair conference management system that –as usual– provided its reliable and useful support of the workshop organization process. Moreover, we would like to thank the members of the Steering Committee of EMAS for their valuable suggestions and support.

April 6, 2016

Matteo Baldoni
Jörg P. Müller
Ingrid Nunes
Rym Wenkstern

Table of Contents

Program Committee

Natasha Alechina	University of Nottingham
Matteo Baldoni	University of Torino
Luciano Baresi	Politecnico di Milano
Cristina Baroglio	University of Torino
Jeremy Baxter	QinetiQ
Ana L. C. Bazzan	Universidade Federal do Rio Grande do Sul
Olivier Boissier	ENS Mines Saint-Etienne
Rafael H. Bordini	FACIN-PUCRS
Lars Braubach	University of Hamburg
Nils Bulling	TU Delft
Rem Collier	UCD
Massimo Cossentino	National Research Council of Italy
Fabiano Dalpiaz	Utrecht University
Mehdi Dastani	Utrecht University
Louise Dennis	University of Liverpool
Virginia Dignum	TU Delft
Jürgen Dix	TU Clausthal
Amal El Fallah Seghrouchni	LIP6 - University of Pierre and Marie Curie
Baldoino Fonseca	Federal University of Alagoas
Aditya Ghose	University of Wollongong
Adriana Giret	Technical University of Valencia
Jorge Gomez-Sanz	Universidad Complutense de Madrid
Sam Guinea	Politecnico di Milano
Christian Guttmann	Institute of Value Based Reimbursement System
James Harland	RMIT University
Vincent Hilaire	UTBM/IRTES-SET
Koen Hindriks	Delft University of Technology
Benjamin Hirsch	Khalifa University
Tom Holvoet	K.U. Leuven
Jomi Fred Hubner	Federal University of Santa Catarina
Michael Huhns	University of South Carolina
Franziska Klügl	Örebro University
Joao Leite	Universidade Nova de Lisboa
Yves Lespérance	York University
Brian Logan	University of Nottingham
Viviana Mascardi	University of Genova
Philippe Mathieu	University of Lille
John-Jules Meyer	Utrecht University
Frederic Migeon	IRIT
Ambra Molesini	Alma Mater Studiourum - Universtà di Bologna
Pavlos Moraitis	Paris Descartes University
Haralambos Mouratidis	University of Brighton
Jörg P. Müller	TU Clausthal

Ingrid Nunes	UFRGS
Juan Pavón	Universidad Complutense de Madrid
Alexander Pokahr	University of Hamburg
Enrico Pontelli	New Mexico State University
Alessandro Ricci	University of Bologna
Ralph Ronnquist	Intendico Pty Ltd
Sebastian Sardina	RMIT University
Valeria Seidita	University of Palermo
Onn Shehory	IBM Haifa Research Lab
Viviane Silva	IBM Research
Guillermo Simari	Universidad Nacional del Sur in Bahia Blanca
Munindar P. Singh	NCSU
Tran Cao Son	New Mexico State University
Pankaj Telang	North Carolina State University
Wamberto Vasconcelos	University of Aberdeen
Jørgen Villadsen	Technical University of Denmark
Gerhard Weiss	University Maastricht
Michael Winikoff	University of Otago
Wayne Wobcke	University of New South Wales
Pinar Yolum	Bogazici University
Neil Yorke-Smith	American University of Beirut
Rym Zalila-Wenkstern	University of Texas at Dallas

Steering Committee

Matteo Baldoni	University of Torino
Rafael H. Bordini	FACIN-PUCRS
Mehdi Dastani	Utrecht University
Jürgen Dix	TU Clausthal
Amal El Fallah Seghrouchni	LIP6 - University of Pierre and Marie Curie
Paolo Giorgini	University of Trento
Jörg P. Müller	TU Clausthal
M. Birna van Riemsdijk	Delft University of Technology
Tran Cao Son	New Mexico State University
Gerhard Weiss	University Maastricht
Danny Weyns	Linnaeus University
Michael Winikoff	University of Otago

Acknowledgements

Matteo Baldoni was partially supported by the *Accountable Trustworthy Organizations and Systems (AThOS)* project, funded by Università degli Studi di Torino and Compagnia di San Paolo (CSP 2014).

Implementing Norms
Why is it so difficult?

Frank Dignum

Utrecht University, The Netherlands F.P.M.Dignum@uu.nl

Abstract. Many people have made implementations of norms or normative systems over the years. However, the implementations differ widely and no uniform methodology to implement normative systems seemed to have been developed. Why is it so difficult to implement norms? Can't we just have a Norms module that can be added to a system? I will discuss these issues and also point to some possible ways forward.

1 Introduction

In [3] we already described some of the issues that come up when trying to implement norms in agent systems. Because the norms influence the behaviour of the agents, they somehow have to be taken into account during the planning and execution of their actions. It is this *influence* relation between the norms and the plans and actions that is difficult to capture. Let me give a simple example. There is a norm that bikes should stop for a red traffic light. In a typical Dutch city like Amsterdam it would be hard to deduce this norm from just looking at the actions of the bikes. So, there is certainly not a one-on-one relation or rule that makes bikes stop whenever the traffic light is red. From own experience it seems that the norm functions as a heuristic that states that when you are on a bike and getting to a red light you have to check whether you can cross and have to stop when there is traffic coming from the sides streets (that you cannot avoid in any way). However, bikes will also stop when there are children already waiting at the traffic light (because you have to give a good example) or when they spot a policeman that could give you a fine when passing a red light. These examples show that the way a norm influences the actions is at least partly situational. One could argue that thus one has to take both the norm and the current situation as input for the influence relation. Theoretically this is correct, but it can be quite difficult to determine how the norm and situation should be combined. Are there just a lot of special situations that give exceptions to a general rule or are there types of situations that each combine in a different way with the norm? This is not very clear and also has not been investigated in a systematic way (although this has, of course, strong links to modeling norms as a kind of default logic).

However, the situation gets even more complex as not only the present situation influences the way norms influence behaviour, but also the social structure of which this situation is a part. E.g. a biker might stop for a red light if everyone

else also stops, but ignores the light if most people do so. The biker also might disregard red lights habitually (at certain crossings or times) if he knows that in the past this never led to any danger. Finally, if the biker is a school teacher riding ahead of a class of children on the bike he will never pass a red light. So, we have to include social status, roles, practices and history in the list of aspects that should be regarded when modeling the influence of norms on behaviour. The list gets longer and longer and also involves more and more complex concepts. Thus it is clear that modeling the influence of norms on behavior can be very complex and will differ depending on which of these aspects are taken into account.

Of course, norms do not exist in isolation. So, besides the above norm there might be a norm that you have to come in time for a meeting with your boss. When you go on the bike to work and are a bit late it might be that you get a conflict between the norm to stop for a red light and being in time for the meeting. In a good logic tradition one could solve this problem by giving a preference order over the norms such that one will be fulfilled and the other violated when they are in conflict. This could be conveniently handled within the norms module. However, life is not as simple as that. Often the preferences of the norms are situation dependent. If I have a meeting with my boss in which I am going to ask a favor (or promotion) I certainly do not want to be late and possibly annoy him before even starting the meeting. Thus I might risk going through red in order to be in time for the meeting. But if I see a policeman near the traffic light I might stop. The reason being, not that I give priority to the traffic norm, but rather that if the policeman is going to fine me, it will take even more time than just waiting for the light! So, at this moment the planning actually influences the norm preference and which norm might be violated. To further complicate matters one might even consider that passing through the red light and being fined one certainly will be late, when you pass through the red light and not get a fine you certainly will be in time and if you stop for the red light you still have a chance of making it in time to the meeting. It is clear that in these more complex cases it does not suffice to just check which norms are applicable at a certain moment and use those as a kind of filter on the potential plans. There is a interdependence between the planning and the norms that does not (always) allow for a one way influence relation (as would be prefered for any deliberation architecture)!

2 Acting with Norms

Given the above arguments one would think it is better not to incorporate norms in agents at all. Things are not as grim as they seem though. Norms have many aspects and not all aspects are equally important in every application. The first step to take when implementing norms should thus be to determine what the function should be of the norms in the system that is developed. The aspects that relate to that function have to be modeled and implemented. In many cases the norms are seen as constraints that (in principle) can still be violated (either

in specific circumstances, randomly in some cases, or based on another clear criteria). In this case one can have a norms module that is used to check all potential plans and orders them based on how many norms they violate and how efficient the plans are.

However, if the norms should lead to emerging behavior as perceived in reality this will not be enough. In that case more elaborate mechanisms should be designed. This will often also necessitate a more complex deliberation cycle of the agents incorporating more social concepts. This is not very easy, because it is not clear on forehand which concepts are needed exactly and there are no architectures that incorporate these concepts in a systematic way. I.e. there are all kinds of extensions of BDI architectures, but often with only one or two new concepts and usually for a very specific purpose or application area.

What is needed is a richer social model in which norms can play a role and agents have all social concepts available. Based on such a rich conceptual model designers can make choices of which parts are needed for their application and what are the consequences of choices they make (both for possible (emerging) behaviour and also for efficiency). Until such a more elaborate social model exists people will have to start from scratch every time they want to implement norms with a slightly different perspective or function in mind. Although this is useful in its own right as all these implementations might support some particular rich social model and give ideas on how to build this, the danger is that people get tired of using norms because they are difficult and try to circumvent the problems in current day systems with very primitive means. This will lead to poorly designed systems that are not well understood and might lead to unforeseen or unwanted emerging behavior. The research that I have started in [1] and [2] should be seen in this light. Although norms do not feature prominently in these papers, they are the underlying motivation to take this broader perspective and start working on a more encompassing social framework. As stated in the vision paper, we hope that other researchers get inspired by this and will join the research.

3 Biography

Dr. Frank Dignum received his PhD in the previous century and has since been working in Swaziland, Portugal and The Netherlands. He is working on social aspects of software agents with applications in serious gaming and social simulations. He is well known for his work on norms and agent communication and lately for the combination of agents and games. His latest research focuses on creating new agent architectures to build agents that operate in real-time environments and have to cooperate with humans and other agents. He has organized many workshops and conferences on the topics and given tutorials at most major conferences and summer schools on them.

References

1. F. Dignum, R. Prada, and G.J. Hofstede. From autistic to social agents. In *AAMAS 2014*, May 2014.
2. V. Dignum, C.M. Jonker, F. Dignum, and R. Prada. Situational deliberation; getting to social intelligence. In *SocialPath workshop*, 2014.
3. Loïs Vanhee, Huib Aldewereld, and Frank Dignum. Implementing norms? In *Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology - Volume 03*, WI-IAT '11, pages 13–16, Washington, DC, USA, 2011. IEEE Computer Society.

Designing and Programming Multiagent Organizations

Jaime Simão Sichman[1]

Laboratório de Técnicas Inteligentes (LTI)
Escola Politécnica (EP)
Universidade de São Paulo (USP)
Av. Prof. Luciano Gualberto, trav. 3, 158
05433-970, São Paulo, SP, Brazil
jaime.sichman@poli.usp.br

Abstract. In the last years, social and organizational aspects of agency have become a major issue in MAS research. Recent applications of MAS on Web Services, Grid Computing and Ubiquitous Computing enforce the need of using these aspects in order to ensure some social order within these systems. One of the ways to assure such a social order is through the so-called *multiagent organizations*. Multiagent organizations are of two types: either the organization emerge from the activity of the individual agents or it is designed to facilitate and guide some specific global behavior. In the latter case, systems are characterized by the autonomy of the individual participants that however must be able to collaboratively achieve predetermined global goals, within a globally constrained environment. However, there is still a lack of a comprehensive view of the diverse concepts, models and approaches related to multiagent organizations. Moreover, most designers have doubts about how to put these concepts in practice, i.e., how to design and how to program them. This invited talk aims to give some possible answers to such questions.

Short Biography

Jaime Simão Sichman is an Associated Professor at University of São Paulo, from where he has obtained both his B.E. and M.E. degrees. He was one of the first students to obtain an European label to his PhD degree, developed at the Institut National Polytechnique de Grenoble (INPG), France, since part of his research was carried out at the Istituto di Psicologia del CNR (currently ISTC), Rome, Italy. He has also spent an abbreviated post-doctoral period at the University of Utrecht, at the Netherlands. His main research focus is multi-agent systems, more particularly social reasoning, organizational reasoning, multi-agent-based simulation, reputation and trust, and interoperability in agent systems. He has advised/co-advised 14 MSc, 12 PhD and several undergraduate students. With other colleagues, he was one of the founders of two subdomains in Multiagent systems, namely Multi-Agent-Based Simulation (MABS) and Coordination, Organization, Institutions and Norms in Agent Systems (COIN), that have originated two successful international workshop series. He has published more than

160 papers in national and international conferences and journals. He is member of the editorial board of the Journal of Artificial Societies and Social Simulation (JASSS), Mediterranean Journal of Artificial Intelligence, Computación y Sistemas, Iberoamerican Journal of Artificial Intelligence, Knowledge Engineering Review (KER) and International Journal of Agent-Oriented Software Engineering (IJAOSE). He has organized several workshops and international conferences and workshops; in particular he was the General Chair (2000) and Program Co-Chair (2006) of the Joint Brazilian/Iberoamerican Conference on Artificial Intelligence (SBIA/IBERAMIA), a member of IJCAI Local Arrangment Committee (2003) and Advisory Committee (2015), the General Chair (2014) of the World Conference on Social Simulation (WCSS) and AAMAS Tutorial Chair (2007), Program Co-Chair (2009) and Local Chair (2017). He was a member of the Brazilian Computer Society (SBC) Advisory Board between 2005 and 2009, and was the coordinator of its Artificial Intelligence Special Commission (CEIA) between 2000 and 2002. He was also the director of the Centro de Computação Eletrônica (CCE) of the University of São Paulo (USP) from 2010 to 2013.

[1] Jaime Simão Sichman is partially financed by CNPq, proc.303950/2013-7.

nDrites: Enabling Laboratory Resource Multi-Agent Systems

Katie Atkinson[1], Frans Coenen[1], Phil Goddard[2], Terry R. Payne[1], and Luke Riley[1,2]

(1) Department of Computer Science, University of Liverpool,
Liverpool L69 3BX, United Kingdom, {atkinson,coenen,payne,l.j.riley}@liverpool.ac.uk.
(2) CSols Ltd., The Heath, Business & Technical Park, Blacon,
Runcorn WA7 4QX, United Kingdom, {phil.goddard,luke.riley}@csols.com.

Abstract. The notion of the multi-agent interconnected scientific laboratory has long appealed to scientists and laboratory managers alike. However, the challenge has been the nature of the laboratory resources to be interconnected, which typically do not feature any kind of agent capability. The solution presented in this paper is that of nDrites, smart agent enablers that are integrated with laboratory resources. The unique feature of nDrites, other than that they are shipped with individual instrument types, is that they possess a generic interface at the "agent end" (with a bespoke interface at the "resource end"). As such, nDrites enable the required interconnectivity for a Laboratory Resource Multi Agent Systems (LR-MAS). The nDrite concept is both formally defined and illustrated using two case studies, that of analytical monitoring and instrument failure prediction.

1 Introduction

Analytical laboratories form a substantial industry segment directed at chemical analysis of all kinds (clinical, environmental, chemical, pharmaceutical, water, food etc). Supplying this marketplace is a $100B per annum industry. Laboratory instruments come in many forms but are broadly designed to undertake a particular type of chemical analysis. Examples of laboratory instrument types include: inductively coupled plasma - mass spectrometers (ICP-MS) for elemental analysis and Chromatography systems for analyte separation. Such laboratory instruments, although usually "front-ended" by a computer resource of some kind, typically operate in isolation. This is because the interfaces used are specific to individual instrument types (of which there are thousands) and individual manufactures. The industry acknowledges that there are significant benefits to be gained if instruments, of all kinds, could "talk" to each other and to other devices [10, 22]; an ability to support remote monitoring/managing of instruments would on its own be of significant benefit. A potential solution is the adoption of a Multi-Agent Systems (MAS) approach to laboratory resource interconnectivity: a Laboratory Resource Multi Agent System (LR-MAS).

However, at present, there is no simple way whereby the LR-MAS vision can be realised. This is not only because of the multiplicity of different interfaces for different models, but also the complex mappings, translations and manipulations that have to be undertaken in order to achieve the desired interconnectivity. Even when just considering specific laboratory instruments, rather than the wider range of laboratory resources,

there are many thousands of models being sold at any one time and a huge variety of legacy systems still in routine use. The limited connectivity that exists is largely focused on what are known as Laboratory Instrument Management Systems (LIMS); systems that receive and store data from instruments (for later transmission to laboratory clients) and manage wider laboratory activities. Some software does exist to facilitate connectivity, for example the L4L (Links for LIMS) software package produced by CSols Ltd[1] (a provider of analytical laboratory instrument software); but this still requires expensive on-site visits by specialist engineers to determine the desired functionality and the nature of the bespoke interfacing. All this serves to prevent the adoption of MAS capabilities within the analytical laboratory industry, despite the general acknowledgement that large scale MAS connectivity will bring many desirable benefits [10, 22].

The technical solution presented here is that of "smart agent enablers" called *nDrites*; an idea developed as part of a collaboration between CSols Ltd and a research team at the University of Liverpool, directed at finding a solution to allow the realisation of the LR-MAS vision. The nDrite concept is illustrated in Figure 1. As shown in the figure, nDrites interact, at the "resource end", in whatever specific way is required by the laboratory resource type in question; whilst at the other end nDrites provide generic interaction. Note that in the figure, for ease of understanding, the nDrite is shown as being separated from the laboratory resource (also in Figure 2), in practice however nDrites are integrated with laboratory resources. Thus nDrites provide system wide communications so as to allow agents to interact with laboratory resources to (say): (i) determine the current state of an entire laboratory system, (ii) determine all past states of the system (system history) or (iii) exert control on the laboratory resources operating within a given laboratory framework. Thus, in general terms, nDrites are a form of intelligent middleware that facilitate LR-MAS operation. The main advantage offered is that of cost. The idea is to build up a bank of nDrites, one per instrument type, that are integrated and shipped with the individual instruments in question. This will then alleviate the need for expensive on-site visits and provide the desired LR-MAS connectivity. The research team already have nDrites in operation with respect to two instrument types (an auto-sampler and an ICP-MS[2]).

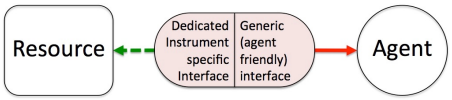


Fig. 1. nDrite smart agent enabler

The main contributions of this paper are thus: (i) the concept of nDrite smart agent enablers that facilitate multi-agent laboratory resource interconnectivity, (ii) the associated formalism that provides for the generic operation of nDrites, and (iii) two case studies illustrating the utility of the nDrite concept (the first currently in production, the second under development). The rest of this paper is organized as follows. In Section 2 some related work is presented. Our Laboratory Resource Multi-Agent System (LR-

[1] `http://www.csols.com/wordpress/`.

[2] The autosampler is manufactured by Teledyne CETAC Technologies, `http://www.cetac.com`, while the ICP-MS is manufactured by Perkin-Elmer, `http://www.perkinelmer.com/`.

MAS) framework, including the nDrite concept, is presented in Section 3. In Section 4, we detail the communication method for our LR-MAS and how nDrites handle communication aspects. The operation of the framework is then illustrated using two nDrite application case studies. The first (Section 5) is an analytical monitoring case study agent, the second (Section 6) is a resource monitoring application agent that operates using a data stream classifier. The paper concludes with some discussion in Section 7.

2 Previous Work

The notion of the pervasive, service rich and interconnected scientific laboratory has long appealed to scientists and laboratory managers of all kinds [10, 22]. Many scientific laboratory processes have traditionally involved using a number of separate, but interconnected tasks, performed by different systems and services (often bespoke) with little support for automated interoperation or holistic management at the laboratory level. To facilitate this interconnectivity, early work was directed at service oriented infrastructures using *Grid* (and later, *Cloud*) computing [8, 9, 15, 24], whereby laboratory equipment, high-performance processing arrays, data warehouses, and *in-silico* scientific modelling was wrapped, and managed, by a service-oriented client [8, 9]. The main focus was that of a "service marketplace" used to discover different services [19] and to schedule or provision their use, as well as to provide support for tasks such as: security [2], notification [17], and scheduling [24]. The need for intelligent, autonomous support for such Grid infrastructures has been well documented [8, 9, 14, 18, 24, inter alia].

The Grid Computing based laboratory infrastructure idea has now been superseded by the emergence, and wide-scale adoption, of Web Services, and consequently MAS, which exploit many of the standards used for the web, and resolved many problems of interoperability between organisations that can effect grid based approaches. This migration was essential to mitigate some of the pragmatic challenges with the interconnection of services within an Open Agent Environment [29]; however, the flexible interoperation of systems and services (developed by different stakeholders with different assumptions) is still a challenge. This motivated the adoption of a wrapper-based approach to support wide spread usability within the nDrite concept.

The laboratory instrument MAS vision thus provides for the automation of process models and workflows [28, 19]; sequences of processes that can occur both serially and in parallel to achieve a more complex task. The laboratory workflow concept has been extensively researched. The fundamental idea is that of a collection of software services, whereby each service is either a process (often semantically annotated [8, 19, 14]), or manages and controls some laboratory resource. Such workflows are typically orchestrated using editors or AI-based planning tools [19], resulting in either an instantiated workflow (one where the specific service instances are identified and used) or in an abstract workflow (one where the instantiation of the services themselves is delayed until execution time). Stein et al. [24, 25] explored the use of an agent-based approach to automatically discover possible service providers where abstract services are defined within a workflow, by using probabilistic performance information about providers to reason about service uncertainty and its impact on the overall workflow. The idea was that by coordinating their behaviours, agents could "re-plan" if the providers of other

services discovered problems in their provision, such as failure, or unavailability. An interesting aspect of this workflow planning approach was the use of autonomously requesting redundant services for particularly critical or failure-prone tasks (thus increasing the probability of success). However, to facilitate the notion of autonomous control, the services themselves need to be endowed with the necessary capabilities to be self monitoring (and thus self aware), discoverable, and communicable [20].

The notion of agents supporting the management of laboratory services through interoperation and workflow (either defined a-priori or dynamically at runtime) is only possible if the agents describe and publish their capabilities, using some discovery mechanism [7]. Although many formalisms (such as UDDI, JINI, etc) have been proposed to support *white* and *yellow* page discovery systems, the discovery of agent-based capabilities based on knowledge-based formalisms describing inputs, outputs, preconditions and effects was pioneered by Sycara et. al. in the work on LARKS [26], and later with the *Profile Model* within OWL-S [1] and the machinery required to discover them [21]. However, before these descriptions and their underlying semantics can be defined, a formal model of the agent capabilities, and their properties should be modelled.

In the above previous work on the automation of process models and workflows using MAS technology, it was assumed that communication services would either be provided by some common or standardised interfaces or through some kind of mediator [27]. However, as noted in the introduction to this paper, there is no agreed communication standard currently in existence, nor is there likely to be so; whilst currently available mediators are limited to bespoke systems such as CSols' L4L system. Hence the nDrite concept as proposed in this paper.

3 The Laboratory Resource Multi-Agent System Framework

A high level view of the proposed nDrite facilitated Laboratory Resource Multi-Agent System (LR-MAS) framework is presented in Figure 2[3], where various laboratory resources are connected to nDrites, including: (i) two laboratory instruments (laser ablation systems, auto-samplers, mass spectrometers, etc.), (ii) a Laboratory Instrument Management System (LIMS) and (iii) a "links for LIMS" system (CSols' legacy mechanism for achieving instrument connectivity to LIMS, but still in operation). The figure also shows two users and a number of agents; for of which are connected directly to one or more nDrites. Two provide linkages between pairs of laboratory resources, and another two others are simply "front ends" to resources. The two remaining agents are application agents, not directly connected to nDrites: one is an Instrument Failure prediction agent and the other an Analytical Monitoring agent. We introduce Ag to denote the set of all possible agents in a LR-MAS, where $Ag = \{ag_1, ag_2, \ldots, ag_n\}$.

As noted in the introduction to this paper the interconnectivity between agents and laboratory resources in our LR-MAS is facilitated by the nDrite smart agent enablers (see Figures 1 and 2). The nDrites can be considered to be wrappers for laboratory resources in the sense that they "wrap" around a laboratory resource to make the laboratory resource universally accessible within the context of a MAS (LR-MAS). As such,

[3] This figure represents a high level vision; in practice the connectivity/operation will be more restrictive for reasons of data confidentiality and business efficacy.

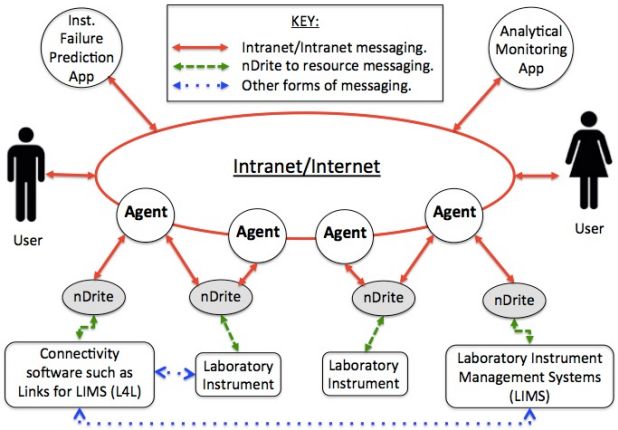


Fig. 2. nDrite facilitated Laboratory Resource Multi-Agent System (LR-MAS) configuration

nDrites can be viewed as being both the agent actuators and sensors for the laboratory resources with which they may be paired. This section provides detail of the nature of nDrites. More specifically, a formalism is presented to enable the LR-MAS vision given above. The section is organised as follows. Sub-sections 3.1 and 3.2 present the formalism with respect to laboratory resources and nDrites (in their role as actuators and sensors), respectively.

3.1 Laboratory Resources

As already noted, individual laboratories comprise a number of laboratory resources. We introduce the set of laboratory resources as $L = \{L_1, L_2, \ldots, L_n\}$. Each laboratory resource has a set of one or more actions that the laboratory resource can perform. The complete set of possible actions that laboratory resources can perform is denoted by $Ac = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$. To find the set of actions an individual resource L_i can perform we use the partial action function $\texttt{LRact} : L \mapsto 2^{Ac}$. Given that there are many different types of laboratory resources (laboratory instruments, robots, data systems, and so on) resources can be grouped into a set of categories $T = \{T_1, T_2, \ldots, T_n\}$, where each T_i is some subset of L ($T_j = \{L_p, L_q, \ldots, L_z\}$). Each category is referred to as a *laboratory resource type*. Thus $\forall T_j \in T$, $T_j \subseteq L$ and $\forall L_i \in T_j$, $L_i \in L$. The intersection of the actions of all laboratory resources of a particular laboratory resource type are called the *critical actions* for that type, denoted $Ac^{\cap T_j}$ where for type T_j: $\bigcap_{\forall L_i \in T_j} \texttt{LRact}(L_i) = Ac^{\cap T_j}$. Note that individual resources can feature other individual actions that are not shared through the critical action set.

3.2 nDrites

The principal function of nDrites is to provide MAS connectivity without exposing the detailed operation of individual laboratory resources of many different kinds and the many different data formats. Recall that laboratory instruments are produced by

many different vendors each using proprietary data formats; there are no standardised language or communication protocols for these different resources. Therefore, nDrites are used as wrappers for laboratory resources to provide a standardised method for communicating data from, and exerting control over, every *nDrite enhanced* laboratory resource. As such, nDrites can be viewed as both actuators and sensors. A formal definition of the operation of nDrites is presented below: initially in the context of nDrites as actuators and later in the context of nDrites as sensors.

nDrites as Actuators The set of nDrites are denoted as $Den = \{D_1, D_2, \ldots, D_n\}$, and the set of possible nDrite actions that the complete set of nDrites Den can expose is $Dc = \{\delta_1, \delta_2, \ldots, \delta_n\}$. The following partial nDrite *action function* defines the set of nDrite actions that a given nDrite can expose $\texttt{DenAct}$: $Den \mapsto 2^{Dc}$. Some nDrite actions may only be possible with respect to particular laboratory resources, others will be *critical actions* shared across a single laboratory resource type or a number of types. To find the set of laboratory resource types to which an nDrite action may be applied we use the function $\texttt{pos} : Dc \mapsto 2^T$. Actions may also be sequenced to define workflows.

Each nDrite action δ_i requires a corresponding *action object*, which details all the necessary parameters for δ_i to operate successfully. The set of nDrite action objects is $Oa = \{oa_1, oa_2, \ldots, oa_n\}$. Each nDrite action object has a class type whereby each object belongs to a class which in turn defines the nature of the object. The set of nDrite object class types is given by $Ot = \{ot_1, ot_2, \ldots, ot_n\}$. The class type of each nDrite action object is found by the following function $\texttt{type}$: $Oa \mapsto Ot$. To find out which class type is required for each nDrite action δ_i, we use the object requirement function $\texttt{req}$: $Dc \mapsto Ot$ (we assume only one object type is required for each nDrite action).

Recall that individual laboratory resources are likely to perform individual actions in different ways. Hence, at the resource end, nDrites have bespoke interfaces (see Figure 1). As such, nDrites are paired with individual laboratory resources (recall Figure 2). An nDrite D_j and a laboratory resource L_i that are connected together are thought of as an *agent enabling pair*: $AEP_k = (L_i, D_j)$. The set of all agent enabling pairs is defined as $AEP = \{AEP_1, AEP_2, \ldots, AEP_n\}$.

Consequently, given an nDrite-laboratory resource pairing, the nDrite functionality can be mapped onto the resource functionality. Additionally, note that an nDrite action δ_i for an nDrite may also include additional software only actions. A software only action is an operation performed internally to the nDrite itself with no engagement with its paired laboratory resource (for example "return the nDrite identification number"). The set of software only actions are $S = \{s_1, s_2, \ldots, s_n\}$. Therefore nDrite actions map onto zero, one or many laboratory resource actions and zero, one or many software only actions[4]. To find the set of laboratory resource and/or software only actions that occur when an nDrite action is called, we use the partial nDrite exposure function $\texttt{exp}$: $Aep \times Dc \mapsto 2^{Ac} \cup 2^S$. Given $L_i \in T_k$ then $\forall \delta_k$ where $T_k \notin \texttt{pos}(\delta_k)$, the following holds: $\texttt{exp}((L_i, D_j), \delta_k) = \emptyset$. That is, zero laboratory resource and software only actions occur when an nDrite action δ_k is attempted to be invoked on an nDrite that cannot perform it. Should an nDrite D_j want to perform an action $ac \in Ac$ on

[4] Note that the number of exposed nDrite actions can therefore be greater than the number of instrument actions.

its paired laboratory resource L_i, then it calls the function $\texttt{Perform}(ac, L_i)$. Should an nDrite D_j want to perform an action $ac \in S$ on itself, then it calls the function $\texttt{Perform}(ac, D_j)$. In both cases a Boolean is returned to indicate whether the action was successful (true) or not (false). We do not describe in detail what occurs in the `Perform` function due to the bespoke interface with the laboratory resource.

To summarise, nDrites can expose all possible actions that a laboratory resource can provide, as well as expose more software only actions. Additionally, nDrites can lower the computational burden for associated agents by exposing sequences of software and laboratory resource actions (workflows). In this manner, nDrites *enhance* the capabilities of the laboratory resources that they are attached to. Of course, for agents to trigger nDrites to perform functions, the agents must know what nDrite actions each nDrite provides. We assume this discovery capability is provided by a yellow pages agent (see [4]), which sits in the LR-MAS (not shown in Figure 2).

nDrites as Sensors For agents to work correctly with nDrites (and therefore the laboratory resources they are connected to), nDrites need to not only be actuators but also sensors. Therefore nDrites map laboratory resource actions into objects that can be understood in our LR-MAS. Previously we mentioned that nDrites, in their actuator role, receive nDrite action objects, which are required for nDrites to perform actions. Concurrently nDrites act as sensors and produce *nDrite sensor objects*. The set of nDrite sensor objects are $Os = \{os_1, os_2, \ldots, os_n\}$. Each object has a class type, where the set of object class types are defined as $Ot = \{ot_1, ot_2, \ldots, ot_n\}$ (note that this is the same definition as object types for nDrite action objects). The type of each sensor object is found by the following function $\texttt{type} : Os \mapsto Ot$. The set of sensor objects that an nDrite maps a set of laboratory resource actions onto, is found using the function $\texttt{Sen}$: $2^{Ac} \times \mathbb{N} \mapsto 2^{Os}$, where the natural number represents the current time point.

Every nDrite D_j collects the nDrite sensor objects it generates in an associated nDrite sensor database (SDB_j)[5] that grows monotonically over time (timepoint $t = 1$ occurs when the nDrite is turned on). Depending on the end users needs, nDrite sensor databases can be local to the nDrite itself, sit on a laboratory server, or be in the cloud. The sensor database is defined as:

Definition 1: **nDrite Sensor database.** *: The database SDB_j for an agent enabling pair (L_i, D_j) holds a set of nDrite objects Osi (where $Osi \subseteq Os$), that have been generated by D_j because L_i has performed the actions LAc (where $LAc \subseteq Ac$).*

$$SDB_i^t = \begin{cases} \emptyset & \textit{iff } t = 0, \\ SDB_i^{t-1} \cup Sen(LAc, t) & \textit{iff } t > 0 \textit{ and } Sen(LAc, t) \neq \emptyset, \end{cases}$$

For nDrites to be sensors for agents, an agent needs to be able to access the objects in the nDrites database. Therefore, included in the software only actions of each nDrite are the following database access functions:

- $\texttt{GetObjectsByOccurances}_i(2^{Ag} \times 2^{Ot} \times \mathbb{N}) \mapsto 2^{Os}$. Returns the most recent n objects of the given object types that occurred in the SDB_i where $n \in \mathbb{N}$.

[5] Additionally there exists an nDrite action database for an nDrite D_j, denoted ADB_j, which holds nDrite action objects.

- $\texttt{SubscribeToObjects}_i(2^{Ag} \times 2^{Ot} \times \mathbb{N})$. Causes $ag \in Ag$ to subscribe to receiving automatic updates concerning sensor objects, saved by nDrite D_i in its database SDB_i, which are of the desired object types, until the given timepoint $n \in \mathbb{N}$.
- $\texttt{UnSubscribeFromObjects}_i(2^{Ag} \times 2^{Ot} \times \mathbb{N})$. Causes $ag \in Ag$ to unsubscribe to receiving automatic updates concerning sensor objects, saved by nDrite D_i in its database SDB_i, which have the desired object types, until the given time point $n \in \mathbb{N}$. If $n = 0$, then the agent is completely unsubscribed.

Additional functions required for the nDrites to operate successfully as agent sensors are as follows:

- $\texttt{GetSubscribers}_i(2^{Ot}) \mapsto 2^{Ag}$. Receives a set of object types and returns the set of agents that have subscribed to these object types.
- $\texttt{GetNextAction}(L \times \mathbb{N}) \mapsto Ac$. Receives a single laboratory resource and a time limit $n \in \mathbb{N}$, and returns the next laboratory action that occurs before the timelimit. If the laboratory resource performs no recognised action within the time limit then $null$ is returned.
- $\texttt{Connected}(2^{Ac} \times 2^{Ac}) \mapsto \{true, false\}$. Returns whether the first set of laboratory resource actions are connected to the second set of laboratory resource actions ($true$) or not ($false$). The two sets are connected if: (i) they form a series that can be converted into an nDrite sensor object; or (ii) they form a series that, when further nDrite sensor objects are added, can be converted into an nDrite sensor object. Also, $true$ is returned if the first set of laboratory resource actions are the empty set. $False$ is returned if the second set of laboratory resource actions are the empty set or if both sets are empty.
- $\texttt{nDriteAdvertisingObjects}_i(2^{Ot}) \rightarrow \{true, false\}$. Returns whether D_i is advertising that it can update the agents on the given set of object types ($true$) or not ($false$). Again, it is assumed that this advertisement is performed using a yellow pages agent.
- $\texttt{CollectSensorObjects}_i(\mathbb{N}) \rightarrow 2^{Os}$. Returns the objects from the database SDB_i that have occurred since the time point $n \in \mathbb{N}$.

4 LR-MAS Communication

So far we have shown that nDrites have the available functionality to be agent actuators and sensors. Note that the nDrite concept isn't simply allowing an agent to perform an action and then observe the result. nDrites allow agents to subscribe to nDrite objects, which may be generated from real world actions (e.g. a user turns an instrument off), or from other agents (e.g. another agent requests the instrument to analyse some samples). Different agents maybe interested in different actions, and so a complicated LR-MAS occurs. As nDrites are separate software entities to agents, there needs to be a communication mechanism available for the agents to utilise the actuator and sensor capabilities of the nDrites. In 4.1, we detail the message syntax between nDrites and agents. Note that the associated message syntax for agent to agent communication is considered to be out of the scope of this paper, however this can clearly be achieved using a FIPA compliant agent communication language. Sub-sections 4.2 and 4.3 show:

(i) how nDrites, in their role of agent actuators, handle incoming messages; and (ii) how nDrites, in their role as agent sensors, produce messages that get sent to agents. Finally, Sub-section 4.4 gives a brief definition for LR-MAS agents.

4.1 nDrite Message Syntax

The LR-MAS given in Figure 2 features a set of communicating entities (agent-nDrite pairs). Messages are sent between these entities, from the set of possible messages, denoted by $M = \{m_1, m_2, \ldots, m_n\}$. Each message contains *meta deta* (denoted by MD), a set of *nDrite actions and nDrite action objects pairs*[6] (NAP, where a single pair is indicated by the tuple $\langle \delta_i, oa_k \rangle$), and a set of *nDrite sensor objects* (NSO). We assume that the meta data must include two functions `Sender` and `Receiver` that returns an entity in either the set of nDrites Den or the set of agents Ag.

Definition 2: *An* **nDrite system message** *is a tuple denoted* $m_i = \langle MD, NAP, NSO \rangle$ *where the following holds:*

1. `Receiver`$(MD) \in Ag \cup Den$
2. *Sender*$(MD) \in Ag \cup Den$
3. *If* `Receiver`$(MD) \in Ag$ *then* `Sender`$(MD) \in Den$
4. *If* `Receiver`$(MD) \in Den$ *then* `Sender`$(MD) \in Ag$
5. *If* $NAP \neq \emptyset$ *then* $\forall \langle \delta_i, oa_k \rangle \in NAP$ *the following holds:*
   (a) $\delta_i \in Dc$; *(b)* $oa_k \in Oa$; *(c)* $oa_k \in$ `req`(δ_i)
6. $NSO \subseteq Os$

Thus an nDrite system message must have a designated receiver and sender (conditions 1 and 2). One out of the sender and receiver one must be an agent, while the other must be an nDrite (conditions 3 and 4). For each nDrite action object pair (NAP), the nDrite action called for must be valid (condition 5(a)), the paired nDrite action object must be valid (condition 5(b)) and the paired nDrite action object must be required by the nDrite action they are paired with (condition 5(c)). Finally, the nDrite sensor objects NSO that are provided must be part of the sensor object set Os (condition 6).

4.2 Sending messages to nDrites

In the *agent actuator* context, the nDrites will have to deal with many incoming messages from agents. In Algorithm 1, we present our general nDrite procedure for dealing with an incoming message. The algorithm starts with the message being unpacked (line 5). Then two sets are initialised, one for the set of nDrite actions that complete successfully (line 6) and another for the nDrite actions that do not complete successfully (line 7). As nDrites are providing wrappers for laboratory resources (instruments, LIMS, etc), an nDrite action can fail through no fault of the nDrite software. For example, a laboratory instrument message could be blocked, or the server that hosts a LIMS could fail. Therefore each nDrite records which actions have succeed and which have failed (so as to help the error recovery process for the agents within our LR-MAS).

[6] nDrite action object pairs are the objects that are saved in the nDrite action database.

Algorithm 1: The `nDriteReceive` algorithm that handles an incoming message for the nDrite D_j that is paired with the laboratory resource L_i.

function `nDriteReceive`(m_i)
Input: $\langle m_i \rangle$; where m_i is the received message.

begin;
$m_i = \langle MD_i, NAP_i, \emptyset \rangle$; // Unpack the message. No sensor objects from agents
$succ = \emptyset$; // Set of successful actions
$fail = \emptyset$; // Set of failed actions
$p = 0$; // Integer count variable for nDrite actions
$t = 0$; // Current timestamp that automatically updates
$complete = false$; // Boolean that notes whether the last action completed or not
$os_j \subset Os$; // nDrite sensor object defined

if `Receiver`$(MD_i) \neq D_j$ **then**
  return $null$; // If this nDrite is not the intended recipient then quit
end if
while $p < |NAP|$ **do**
  $\langle \delta, oa_k \rangle_p \in NAP$;
  if $\delta \in$ `DenAct`(D_j) **and** $(oa_k =$ `req`$(\delta))$ **then**
    $q = 0$; // Integer count variable for individual actions
    $ADB_j^t = ADB_j^{t-1} \cup \langle \delta, oa_k \rangle_p$;
    while $q <$ |`exp`$((L_i, D_j), \delta)|$ **do**
      $ac_q \in$ `exp`$((L_i, D_j), \delta)$;
      if $ac_q \in S$ **then**
        $complete =$`Perform`(ac_q, D_j); // I.e. ac_q is a software only action
      else
        $complete =$`Perform`(ac_q, L_i); // I.e. ac_q is a laboratory resource action
      end if
      if $complete = true$ **then**
        $\langle \delta_i$, error information$\rangle \in fail$;
      else
        $\langle \delta$, success information$\rangle \in succ$;
      end if
      $q++$;
    end while
  else
    $\langle \delta_i$, error information$\rangle \in fail$;
  end if
  $p++$;
end while
$fail, succ \in os_j$; // Add the success and fail information to an nDrite sensor object
$m_j = \langle MD_j, \emptyset, \{os_j\} \rangle$; // Add sensor object to return message
`Receiver`$(MD_j) =$ `Sender`(MD_i);
`Sender`$(MD_j) =$ `Receiver`(MD_i);
Send m_j;
end;

The first thing an nDrite should check when a message is received, is whether it was the intended receiver (line 13 in Algorithm 1). If it was not the intended receiver the message is ignored (line 14), otherwise the message is processed (line 16 onwards). When processing the message, the nDrite takes one nDrite Action-object Pair ($\langle \delta, oa_k \rangle$) at a time (line 17). If this nDrite can perform the required nDrite action δ, and the required nDrite action object has been received (line 18), then δ is processed. Whenever an nDrite action object pair is to be processed, this is saved into the nDrite action database (line 20), so that a record of the system history is available. The nDrite processes δ by converting it into a sequence of laboratory resource and software actions via the `exp` function (line 20). If the next action ac_q is a software action, then it is performed on the nDrite (line 24), otherwise it is performed on the laboratory resource (line 26). The boolean $complete$ stores details on whether ac_p completed successfully. If any of the actions from the `exp` function are unsuccessful, then the original nDrite action δ (and information on the error) are added to the list of nDrite actions that failed (line 29), otherwise the original nDrite action δ is added to the list of nDrite actions that succeeded (line 31). This process continues until all the nDrite actions in the NAP set have been dealt with (line 16)[7]. Finally, the nDrite builds and sends a message m_j to inform the agent of what actions succeeded and what failed (lines 40 to 44).

4.3 Sending Messages to Agents

In the context of nDrites operating as sensors for agents, Algorithm 2 presents the general nDrite sensor algorithm. The algorithm takes as input the laboratory resource L_i that is pared with the nDrite D_j. Therefore the agent-enabling pair is set as (L_i, D_j). The algorithm begins by launching a database monitoring thread (line 9), the purpose of which is to monitor this nDrite's sensor database and send updates to the subscribing agents once sensor objects of the correct type appear in the database (this thread is described in more detail later). The main function then processes sequences of laboratory resource actions (describing a workflow) until termination (line 10). The Φ variable holds the current laboratory resource action series (workflow) that is being recorded[8]. This action series is initially set to empty (line 8).

When processing an action series (workflow) the first laboratory resource action is added to the current laboratory action series, as the `Connected` function always returns true when the current series is empty (line 12 and 13). Next the nDrite checks whether it advertises that it can update agents on the nDrite sensor objects that would appear from the conversion of the current action series (line 23). If so, these converted objects are added to the nDrite's sensor database (line 24), as monitored through the `nDriteMonitorDB` function. Next the nDrites waits until $timelimit$ for the next

[7] Note that if the instrument is currently busy, then the `perform` function will return false and the agent will be alerted through the error information stored in $fail$.

[8] A laboratory resource action sequence (workflow) can be processed by the nDrite as a collection; for example a sample analysis by a laboratory instrument. Single instrument actions can be: move to the next sample; send this sample for analysis; record sample results; move to next sample; etc. Some of this information maybe useful to some agents who want real time updates but other agents maybe "happy" to just have information on a collection of actions.

Algorithm 2: The `nDriteMonitor` algorithm allows the nDrite D_j to monitor the laboratory resource L_i and convert any laboratory resource actions into LR-MAS understandable nDrite sensor objects. Once converted, the nDrite will update any agents that have subscribed to these nDrite sensor object types.

function `nDriteMonitor(`L_i`)`
Input: $\langle L_i \rangle$; where L_i is the Laboratory resource to monitor.

begin;
$p = 0$; // Integer count variable for nDrite action
$t = 0$; // Current timestamp that automatically updates
$timelimit$ // A predefined integer to wait for the next lab resource action
$\Phi = \emptyset$; // Laboratory action series initialised
start `nDriteMonitorDB()` in new thread
while nDrite not terminated **do**
  $\alpha_p =$ `GetNextAction`$(L_i, timelimit)$
  if `Connected`$(\Phi, \{\alpha_p\})$ **then**
    $\Phi_p = \alpha_p$; // Action is added to action series
    $p++$;
  else if $\alpha_p \neq null$ **then**
    $\Phi = \emptyset$; // This action series has ended
    $\Phi_0 = \alpha_p$; // A new action series is initialised with the last action
    $p = 1$;
  else
    $\Phi = \emptyset$; // This action series has ended
    $p = 0$;
  end if
  if `nDriteAdvertisingObjects(type(Sen(`Φ, t`)))` **then**
    $SDB_j^t = SDB_j^{t-1} \cup$ `Sen(`Φ, t`)`;
  end if
end while
end;

function `nDriteMonitorDB()`
begin;
Integer $s = 0$; // Last timestamp checked
while nDrite not terminated **do**
  $\Gamma =$ `CollectObjects(`s`)`;
  $s =$ current time;
  for each $os_i \in \Gamma$ **do**
    for each $ag_j \in$ `Subscribers(type(`os_i`))` **do**
      $m_k = \langle MD, \emptyset, \{os_i\} \rangle$;
      `Sender(`MD`)` $= D_j$; `Receiver(`MD`)` $= ag_j$;
      send m_k;
    end for
  end for
end while
end;

laboratory resource action in the series occurs (line 11). If it does not occur before $timelimit$ then the laboratory resource action will be set to $null$ (the current workflow has been completed), so `Connected` will return $false$ (line 12) and the actionSequence will be broken (line 20 and 21). Conversely if another laboratory action is found within the time limit (line 11), then if `Connected` returns true, the new action α_p is added to the sequence Φ and the process continues (lines 13 and 14). If `Connected` returns false, then α_p is not added to the current sequence, which completes (line 16), and instead, α_p becomes the first action of a new sequence (lines 17 and 18).

The `nDriteMonitorDB` thread continues to run until the nDrite terminates. The first part of the continuous loop collects nDrite sensor objects into Γ, which have occurred in this nDrite's database since the last time it checked (line 33). The last check time is then updated (line 34). For every nDrite sensor object os_i found (line 35), and for each agent ag_j that subscribes to updates concerning the objects of the type `type`(os_i) (line 36), a message is sent to each agent ag_j to inform it of the update (lines 37 to 39).

4.4 Definition of LR-MAS Agents

As discussed, there are extensive possibilities for LR-MAS agents, so we make no assumptions regrading their structure. At a highlevel, LR-MAS agents are defined as:

Definition 3: *An* **nDrite enabled LR-MAS agent** *is an autonomous software component that:*

- *Takes as input messages of the form* $\langle MD, NAP, NSO \rangle$
- *Sends messages of the form* $\langle MD, NAP, NSO \rangle$

How agents interpret nDrite sensor objects, and why they would build nDrite action objects is entirely up to them. Individual agents can perform a variety of tasks limited only by the nDrite actions implemented. The current classes of agent focused on for production are: (i) Discovery Agents, (ii) System Configuration Agents, (iii) Analytical Monitoring Agents and (iii) Instrument Monitoring Agents. We now provide two real world examples of nDrite usage (the two App agents in Figure 2). As Figure 2 shows, these two agents can be present in the same LR-MAS and connect to the same nDrites.

5 The Analytical Monitoring Case Study (Case Study 1)

Our first case study is focused on the "AutoDil agent" currently in operation (Figure 2). AutoDil uses two nDrites: (i) an Inductively Coupled Plasma Mass Spectrometer (ICP-MS) nDrite, denoted D_{icp}, and (ii) an autosampler nDrite[9], denoted D_{as}. The purpose of the AutoDil agent is to ensure any samples from the autosampler found to be "over-range" by the ICP-MS instrument are rediluted and sent for reanalysis. An ICP-MS analyses many samples, one after the other. A collection of samples is know as a run. When a run has been completed many laboratory resource actions have been performed, which are converted by the ICP-MS nDrite D_{icp} (through D_{icp}'s `nDriteMonitor` function), into a run results nDrite sensor object os_{rx} of the type ot_{rr}.

[9] An autosampler automatically feeds a liquid sample into an ICP-MS.

For the AutoDil agent ag_{ad} to do its job, it must subscribe to nDrite sensor objects of the type ot_{rr} from the ICP-MS instrument nDrite D_{icp}. Note that D_{icp} will have advertised that it can update agents with respect to objects of the type ot_{rr}, thus $\texttt{nDriteAdvertisingObjects}(\{ot_{rr}\}) = true$. When ag_{ad} receives an nDrite sensor object os_{ry} of type ot_{rr}, then it should analyse os_{ry} to see if any samples in the results run need redilution. Whenever ag_{ad} finds samples that require redilution, it:

1. Builds an nDrite action object oa_x that includes information on the dilution amounts for each sample and calls the `AddDilutions` nDrite action in D_{as} by constructing the message $m_p = \langle MD, \langle\, \texttt{AddDilutions}, oa_x \,\rangle, \emptyset\rangle$, where $\texttt{Receiver}(m_p) = D_{as}$ and $\texttt{Sender}(m_p) = Ag_{ad}$.
2. Builds an nDrite action object oa_y that includes information on which samples to be reanalysed and calls the `SetupRun` nDrite action in D_{icp} by constructing the message $m_p = \langle MD, \langle\, \texttt{SetupRun}, oa_j \,\rangle, \emptyset\rangle$, where $\texttt{Receiver}(m_p)$ D_{icp} and $\texttt{Sender}(m_p) = Ag_{ad}$.

After performing both (1) and (2), the ag_{ad} waits for new objects from the ICP-MS nDrite, which may including information on samples that required further dilution.

The nDrites will deal with messages (1) and (2) through their `nDriteReceive` function. The nDrite D_{as} will convert the nDrite action `AddDilutions` through the `exp` function, to actions that its paired autosampler can understand. The purpose of these converted actions will be to tell the autosampler which samples require what level of dilution. The nDrite D_{icp} will convert the nDrite action `SetupRun`, again through the `exp` function, to actions that its paired ICP-MS instrument can understand. The purpose of these actions will be to tell the ICP-MS instrument what samples it should load from the autosampler (and therefore what data it will be collecting). The nDrites will then report to ag_{ad} what actions were successful. If all were successful then the autoDil agent knows that it should soon expect another run results nDrite sensor object os_{rz} of type ot_{rr}, which will hold information on the diluted samples.

6 The Instrument Failure Prediction Case Study (Case Study 2)

The second case study is an instrument failure prediction scenario where a dedicated agent (see Figure 2) is used to predict instrument failure using a data stream classifier trained for this purpose (as proposed in [3]). This agent is currently under development. Instrument failure within analytic laboratories can lead to costly delays and compromise complex scientific workflows [23]. Many such failures can be predicted by learning a failure prediction model using some form of data stream mining, which is concerned with the effective, real time, capture of useful information from data flows [11–13]. A common application of data stream mining is the analysis of instrument (sensor) data with respect to some target objective [5, 6]. There is little work on using data stream mining to predict the failure of the instruments (sensors) themselves other than [3] which describes a mechanism whereby data stream mining can be applied to learn a classifier with which to predict instrument failure. In our LR-MAS, an instrument failure prediction app agent implements the mechanism of [3] by communicating with other agents that are connected to nDrites (referred to as Dendrites in [3]).

7 Conclusions

We have described a mechanism to realise the benefits of MAS in the context of analytical laboratories where laboratory resources are not readily compatible with the technical requirements of MAS. Our solution is the concept of nDrites, "smart agent enablers", that at one end feature bespoke laboratory resource connectivity while at the other end feature a generic interface usable by agents of all kinds. The vision is that of a Laboratory Resource MAS (LR-MAS). The operation of nDrites was fully described in the context of: laboratory resources, nDrites as agent actuators, nDrites as sensors, the communication mechanisms and the associated agents. The utility of nDrites was illustrated in two case studies: (i) an analytical monitoring case study for an "AutoDil agent" currently in operation; and (ii) a instrument failure prediction case study, featuring monitoring agents, that is currently under development. We believe that the proposed nDrite concept will enable the interconnected scientific laboratories of the future.

Acknowledgements

This work was conducted as part of the "Dendrites: Enabling Instrumentation Connectivity" Innovate UK funded knowledge transfer partnership project (KTP009603).

References

1. Ankolekar, A., Burstein, M., Hobbs, J. R., Lassila, O., Martin, D. L., McDermott, D., McIlraith, S. A., Narayanan, S., Paolucci, M., Payne, T. R. and Sycara, K. *DAML-S: Web Service Description for the Semantic Web.* Proc. of ISWC, 2002.
2. Ashri, R., Payne, T. R., Luck, M., Surridge, M., Sierra, C., Aguilar, J. A. R. and Noriega, P. *Using Electronic Institutions to secure Grid environments.* 10th International Workshop on Cooperative Information Agents. p461-475, 2006.
3. Atkinson, K., Coenen, F., Goddard, P., Payne, T and Riley, L. *Data Stream Mining with Limited Validation Opportunity: Towards Instrument Failure Prediction.* 17th Int'l Conference on Big Data Analytics and Knowledge Discovery, Springer LNCS, p283-295, 2015.
4. Bellifemine, F. L., Caire, G. and Greenwood, D. *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)* John Wiley & Sons, 2007.
5. Cohen, I., Goldszmidt, M., Kelly, T., Symons, J. and Chase, J.S. *Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control.* Proc 6th Symposium on Operating Systems Design and Implementation, p231-244, 2004.
6. Cohen, L., Avrahami-Bakish, G., Last, M., Kandel, A., and Kipersztok, O. *Real Time Data Mining-Based Intrusion Detection.* Information Fusion, 9(3), p344-354., 2008.
7. Decker, K., Sycara, K. and Williamson, M. *Middle-agents for the Internet.* 15th International Joint Conference on Artificial Intelligence (IJCAI'97), p578-583, 1997.
8. De Roure, D., Jennings, N.R. and Shadbolt, N. *The Semantic Grid: A Future e-Science Infrastructure.* Grid Computing-Making the Global Infrastructure a Reality, p437-470, 2003.
9. Foster, I., Jennings, N. R. and Kesselman, C. *Brain meets Brawn: why Grid and Agents need each other.* In Proc of. AAMAS, p8-15, 2004.
10. Frey, J.G., De Roure, D., schraefel, M.C., Mills, H., Fu, H., Peppe, S., Hughes, G., Smith, G. and Payne, T. R. *Context Slicing the Chemical Aether.* 1st International Workshop on Hypermedia and the Semantic Web, Nottingham, UK, 2003.

11. Gaber, M. M., Zaslavsky, A. and Krishnaswamy S. *Mining Data Streams: A Review.* ACM SIGMOD Record, 34(2), p18 - 26, 2005.
12. Gaber, M. M., Gama, J., Krishnaswamy, S., Gomes, J.B. and Stahl, F. *Data Stream Mining in Ubiquitous Environments: State-of-the-Art and Current Directions.* Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery; 4(2), p116-138, 2014.
13. Gama, J (2010). *Knowledge Discovery from Data Streams.* Chapman and Hall.
14. Gil, Y. *From data to knowledge to discoveries: Artificial intelligence and scientific workflows.* Scientific Programming 17(3): p231-246, 2009.
15. Hamdaqa, M. and Tahvildari, L. *Cloud Computing Uncovered: a Research Landscape.* Advances in Computers 86: p41-85, 2012.
16. Jacyno, M., Bullock, S., Geard, N., Payne T. R., and Luck, M. *Self-Organising Agent Communities for Autonomic Resource Management.* Adaptive Behaviour Journal. 21 (1), p3-28, 2013.
17. Lawley, R., Luck, M., Decker, K., Payne, T. R. and Moreau, L. *Automated Negotiation Between Publishers and Consumers of Grid Notifications.* Parallel Processing Letters, 13 (4). p537-548, 2003.
18. Merelli, E., Armano, G., Cannata, N., Corradini, F., d'Inverno, M., Doms, A., Lord, P., Martin, A., Milanesi, L., Moller, S., Schroeder, M., Luck, M. *Agents in Bioinformatics, Computational and Systems Biology.* Briefings in Bioinformatics, 8(1), p45-59, 2007.
19. Oinn T., Greenwood M., Addis, M., Alpdemir, M. N., Ferris, J., Glover, K., Goble, C., Goderis, A., Hull, D., Marvin, D., Li, P., Lord, P., Pocock, M. R., Senger, M., Stevens, R., Wipat, A., and Wroe, C. *Taverna: Lessons in Creating a Workflow Environment for the Life Sciences.* Concurrency and Computation: Practice and Experience, 18(10), p1067-1100, 2006.
20. Payne, T. R. *Web Services from an Agent Perspective.* IEEE Intelligent Systems, 23(2), 2008.
21. Paolucci, M., Kawamura, T., Payne, T. R. and Sycara, K. *Semantic Matching of Web Services Capabilities.* Proceedings of the 1st International Semantic Web Conference (ISWC), 2002
22. Schraefel, M. C., Hughes, G., Mills, H., Smith, G., Payne, T. and Frey, J. *Breaking the Book: Translating the Chemistry Lab Book into a Pervasive Computing Lab Environment.* SIGCHI Conference on Human Factors in Computing Systems, April 24-29, Vienna, Austria, 2004.
23. Stein, S., Payne, T.R. and Jennings, N.R. *Flexible QoS-Based Service Selection and Provisioning in Large-Scale Grids.* UK e-Science All Hands Meeting, HPC Grids of Continental Scope, 2008.
24. Stein, S., Payne, T. R. and Jennings, N. R. *Flexible Selection of Heterogeneous and Unreliable Services in Large-Scale Grids.* Philosophical Transactions of the Royal Society A: Mathematical, Physical & Engineering Sciences, 367 (1897). p2483-2494, 2009.
25. Stein, S., Payne, T. R. and Jennings, N. R. *Robust Execution of Service Workflows using Redundancy and Advance Reservations.* IEEE Trans. Services Computing. 4(2), 2011.
26. Sycara, K., Widoff, S., Klusch, M. and Lu, J. *LARKS: Dynamic Matchmaking Among Heterogeneous Software Agents in Cyberspace.* AAMAS, 5(2), 173-203, 2002.
27. Szomszor, M., Payne, T. R. and Moreau, L. *Automated Syntactic Medation for Web Service Integration.* In: IEEE International Conference on Web Services, Chicago, USA, 2006.
28. Wassink, I., Rauwerda, H., Vet, P., Breit, T., and Nijholt, A. *E-BioFlow: Different Perspectives on Scientific Workflows.* Bioinformatics Research and Development: Second International Conference, Vienna, Austria, July 7-9, 2008.
29. Wens, D., Michel, F. *Agent Environments for Multi-agent Systems - A Research Roadmap.* Agent Environments for Multi-Agent Systems IV: 4th International Workshop, E4MAS 2014 - 10 Years Later, p3-21, 2014.

Data and Norm-aware Multiagent Systems for Software Modularization (Position Paper)

Matteo Baldoni[1], Cristina Baroglio[1], Diego Calvanese[2], Roberto Micalizio[1], and Marco Montali[2]

[1] Università degli Studi di Torino — Dipartimento di Informatica
c.so Svizzera 185, I-10149 Torino (Italy)
{firstname.lastname}@unito.it
[2] Free University of Bozen-Bolzano — KRDB Research Centre
Piazza Domenicani 3, I-39100 Bolzano, Italy c.so Svizzera 185, I-10149 Torino (Italy)
lastname@inf.unibz.it

Abstract. This work surveys the key proposals to the modularization of software, and trace them back to the common ground provided by Meyer's three forces of computation: processor, object, and action. We advocate that a paradigm should provide a good balance in exploiting *all* such forces, and support this stance by explaining the weaknesses of the examined proposals. Then, we focus on the agent paradigm because it emerges as pivotal for the achievement of a good balance. We trace directions that we think should be followed in order to complete the model, identifying, in particular, in data-awareness jointly with a norm-based representation of how data evolution is governed the key advancements that would bring to fullness the modularization of software.

1 Introduction

Research on agents and multiagent systems introduced many abstractions and tools to help designing modularized software, e.g., organizations, interaction protocols, artifacts, norms. This work provides a wide and systematic account of the major approaches to modularization, that were developed both by research on multiagent systems and by other research communities, leveraging Meyer's three forces of computation [31] as reference dimensions, along which all the considered proposals are positioned. The aim of this survey is to identify the lacks of the state of art together with possible directions of research. The paper is so organized. Section 2 introduces Meyer's forces of computation. Section 3 shows how functional decomposition, object-orientation, the actor model, business processes, artifact-centric approaches can be seen as manifestations of either the processor force or of the object force. Section 4 explains the strengths and the lacks of proposals from research area on agents. Section 5 explains the value of the action force, considered as ancillary by most of the examined approaches. Section 6 traces as open directions of research data and information-awareness

jointly with an extended norm-based representation that includes rules that govern the environment. Conclusions end the paper.

2 Meyer's forces: Processor, Action and Object

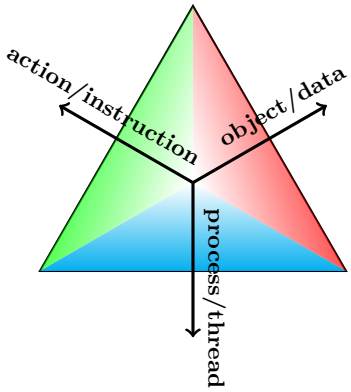


Fig. 1. Meyer's three forces of computation [31, Chapter 5, page 101].

The goal of software engineering is the production of quality software [31]. Among the desired qualities, *correctness* is the ability of software products to perform their tasks as defined by their specification; *robustness* is the ability to react appropriately to abnormal conditions; *extensibility* is the ease of adapting software products to changes of specification; *reusability* is the ability of software elements to serve for the construction of many different applications. In order for software to show these properties, it is necessary to identify proper *modularization mechanisms* that allow the programmer to design and develop software in a systematic way. To evaluate a modularization mechanism, one should not only consider how easy it is, by adopting it, to obtain a software module from scratch, but also how easy it is to maintain that software over time. We decided to use *Meyer's forces of computation* as a common ground for comparing the different proposals because they provide a neutral touchstone, unrelated to any specific programming approach or modularization mechanism. According to Meyer, three forces are at play when we use software to perform some computations (see Figure 1): *processors*, *actions*, and *objects*. A processor can be a *process* or a *thread* (in the paper we use both the terms processor and process to refer to this force); actions are the *operations* that make the computation; objects are the *data* to which actions are applied.

A software system, in order to execute, uses processes to apply certain actions to certain objects. The form of the actions depends on the considered level

of granularity: they can be instructions of the programming language as well as they can be major steps of a complex algorithm. Moreover, the form of actions conditions the way in which processes operate on objects. Some objects are built by a computation for its own needs and exist only while the computation proceeds; others (e.g., files or databases) are external and may outlive individual computations. In the following we analyse the most important proposals concerning software modularization, showing how they (sometimes implicitly) give more or less strength to Meyer's forces, and the drawbacks that follow.

3 Processor vs. Object: the big fight

It becomes apparent that processor and object are the two principal forces along which most approaches to modularization have been developed so far, while the action force remained subsidiary to one or another.

Functional Decomposition. The top-down *functional decomposition* is probably the earliest approach to building modularized software; it relies on a model that puts at the center the notion of process; namely, the implementation of a given function is based only on a set of actions made of instructions, provided by the programming language at hand, possibly in combination with previously defined functions [31]. Top-down functional decomposition builds a system by stepwise refinement, starting with the definition of its abstract function. Each refinement step decreases the abstraction of the specification. With reference to Figure 1, the approach disregards objects/data, just considered as data structures that are instrumental to the function specification and internal to processes. Actions are defined only in terms of the instructions provided by the programming language and of other functions built on top of them (subroutines), into which a process is structured. All in all, this approach is intuitive and suitable to the development of individual *algorithms*, in turn aimed at solving some specific *task*, but does not scale up equally well when *data are shared among concurrent processes* because it lacks abstractions to explicitly account for such data and their corresponding management mechanisms.

Object-Orientation. The *Object-Oriented* approach to modularization results from an effort aimed at showing the limits of the functional approach [31]. Objects (data) often have a life on their own, independent from the processes that use them. Objects become, then, the fundamental notion of the model. They provide the actions by which (and only by which) it is possible to operate on them (*data operations*). This approach, however, disregards processes and their modularization both internally and externally to objects. Internally, because objects provide actions but have a *static nature*, and are inherently passive: actions are invoked on objects, but the decision of which operations to invoke so as to evolve such objects is taken by external processes. This also implies that there is no decoupling between the *use of an object* and the *management of that object*.

Externally, because the model does not supply conceptual notions for composing the actions provided by objects into processes, and there is no conceptual support to the specification of tasks, in particular when concurrency is involved.

Actor Model, Active Objects. The key concept in the *actor model* [28] (to which *active objects* are largely inspired) is that *everything is an actor*. Interaction between actors occurs only through *direct asynchronous message passing*, with no restriction on the order in which messages are received. An actor is a computational entity that, in response to an incoming message, can: (1) send a finite number of messages to other actors; (2) create a finite number of new actors; (3) designate the behavior to be used in response to the next incoming message. These three steps can be executed in any order, possibly in parallel. Recipients of messages are identified by opaque addresses. Interestingly, in [28] Hewitt et al. state that "We use the ACTOR metaphor to emphasize the inseparability of control and data flow in our model. Data structures, functions, semaphores, monitors, [...] and data bases can all be shown to be special cases of actors. All of the above are objects with certain useful modes of behavior." The actor model *decouples* the sender of a message from the communications sent, and this makes it possible to tackle asynchronous communication and to define control structures as patterns of passing messages.

Many authors, such as [32, 44, 35], noted that the actor model does not address the issue of *coordination*. Coordination requires the possibility for an actor to have expectations on another actor's behavior, but the mere asynchronous message passing gives no means to foresee how a message receiver will behave. For example, in the object-paradigm methods return the computed results to their callers. In the actor model this is not granted because this simple pattern requires the exchange of two messages; however, no way for specifying patterns of message exchanges between actors is provided. The lack of such mechanisms hinders the verification of properties of a system of interacting actors. Similar problems are well-known also in the area that studies enterprise application integration [1] and service-oriented computing [43], that can be considered as heirs of the actor model and where once again interaction relies on asynchronous message passing. There are in the literature proposals to overcome these limits. For instance for what concerns the actor model. [35] proposes to use Scribble protocols and their relation to finite state machines for specification and runtime verification of actor interactions. Instead, in the case of service-oriented approaches, there are proposals of languages that allow capturing complex business processes as service compositions, either in the form of orchestrations (e.g. BPEL) or of choreographies (e.g. WS-CDL).

The above problem can better be understood by referring to Meyer's forces. The actor model supports the realization of object/data management processes (these are the internal behaviors of the actors, that rule how the actor evolves), but it does not support the design and the modularization of processes that perform the object use, which would be *external* to the actors. As a consequence, generalizing what [15] states about service-oriented approaches, the modularization supplied by the actor model, while favoring component reuse, does not

address the need of connecting the data to the organizational processes: data remains hidden inside systems.

Business Processes. Business processes have been increasingly adopted by enterprises and organizations to conceptually describe their dynamics, and those of the socio-technical systems they live in. Modern enterprises [14] are complex, distributed, and aleatory systems: complex and distributed because they involve offices, activities, actors, resources, often heterogeneous and geographically distributed; aleatory because they are affected by unpredictable events like new laws, market trends, but also resignations, incidents, and so on. In this light, *business processes* help to create an explicit representation of how an enterprise works towards the accomplishments of its tasks and goals. More specifically, a business process describes how a set of interrelated activities can lead to a precise and measurable result (a product or a service) in response to an *external event* (e.g., a new order) [47]. Business processes developed for understanding how an enterprise work can then be refined and used as the basis for developing software systems that the enterprise will adopt to concretely support the execution of its procedures [14, 25]. In this light, business processes become *workflows* that connect and coordinate different people, offices, organizations, and software in a compound flow of execution [1]. Among the main advantages of this process-centric view, the fact that it enables analysis of an enterprise functioning, it enables comparison of business processes, it enables the study of compliance to norms (e.g. [27]), and also to identify critical points like bottlenecks by way of simulations (e.g., see iGrafx Process[3] for Six Sigma). The adoption of a service-oriented approach and of web services helps implementing workflows that span across multiple organizations, whose infrastructures may well be heterogeneous and little integrated [1, 43].

On the negative side, business processes, by being an expression of the process force, show the same limits of the functional decomposition approach. Specifically, they are typically represented in an activity-centric way, i.e., by emphasizing which flows of activities are acceptable, without providing adequate abstractions to capture the data that are manipulated along such flows. Data are subsidiary to processes.

Artifact-centric Process Management. The *artifact-centric approach* [7, 20, 15] counterposes a data-centric vision to the activity-centric vision described above. *Artifacts* are concrete, identifiable, self-describing chunks of information, the basic building blocks by which business models and operations are described. They are business-relevant objects that are created and evolve as they pass through business operations. They include an *information model* of the data, and a *lifecycle model*, that contains the key states through which the data evolve, together with their transitions (triggered by the execution of corresponding tasks). A change to an artifact can trigger changes to other artifacts, possibly of a different type. The lifecycle model is not only used at runtime to track the evolution

[3] `http://www.igrafx.com/`.

of artifacts, but also at design time to understand who is responsible of which transitions.

On the negative side, like in the case of the actor model, business artifacts disregard the design and the modularization of those processes that operate on them. Moreover, verification problems are much harder to tackle than in the case where only the control-flow perspective is considered. In fact, the explicit presence of data, together with the possibility of incorporating new data from the external environment, makes these systems infinite-state in general [15].

4 Towards Reconciliation: Agents and the A&A meta-model

In [40, 49], *agents* are defined as entities that observe their environment and act upon it so as to achieve their own goals. Two fundamental characteristics of agents are *autonomy* and *situatedness*. Agents are autonomous in the sense that they have a sense-plan-act deliberative cycle, which gives them control of their internal state and behavior; autonomy, in turn, implies proactivity, i.e., the ability of an agent to take action towards the achievement of its (delegated) objectives, without being solicited to do so. Agents are situated because they can sense, perceive, and manipulate the environment in which operate. The environment could be physical or virtual, and is understood by agents in terms of (relevant) data. From a programming perspective, it is natural to compare agents to objects. Agent-oriented programming was introduced by Shoham as "a specialization of *object-oriented programming*" [41]. The difference between agents and static objects is clear. Citing Wooldridge [49, Section 2.2]: (1) objects do not have control over their own behavior[4], (2) objects do not exhibit flexibility in their behavior, and (3) in standard object models there is a single thread of control, while agents are inherently multi-threaded. Similar comments are reported also by other authors, like Jennings [29]. However, when comparing agents to actors, the behavioral dimension is not sufficient: [49, page 30] reduces the difference between agents and active objects, which encompass an own thread of control, to the fact that "active objects are essentially agents that do not necessarily have the ability to exhibit *flexible* autonomous behavior". In order to understand the difference between the agent paradigm and objects it is necessary to rely on both the abstractions introduced by the agent paradigm, that are that of agent and that of environment [48]. Such a dichotomy does not find correspondence in the other models and gives a first-class role to both Meyer's process and object force (see Figure 2). Processes realize algorithms aimed at achieving objectives, and this is exactly the gist of the agent abstraction and the rationale behind its proactivity: agents exploit their deliberative cycle (as control flow), possibly together with the key abstractions of belief, desire, and intention (as logic), so as to realize algorithms, i.e., processes, for acting

[4] This is summarized by the well-known motto "Objects do it for free; agents do it because they want it".

in their environment to pursue their goals[5]. Contrariwise, active objects and actors do not have goals nor purposes, even though their specification includes a process. As we said, they are a manifestation of the object force. In the agent paradigm the manifestation of the object force is the environment abstraction. The environment does not exhibit the kind of autonomy explained for agents even when its definition includes a process. Its being reactive rather than active makes the environment more similar to an actor whose behavior is triggered by the messages it receives, that are all served indistinctly.

Most of the research in multiagent systems typically focuses on the abstraction of agent only, completely abstracting away from the notion of environment. Proposals like [23, 48] overcome this limit by introducing first-class abstractions for the environment, to be captured alongside agents themselves. In particular, [48] states that "the environment is a first-class abstraction that provides the surrounding conditions for agents to exist and that mediates both the interaction among agents and the access to resources." This proposal brought to important evolutions like the A&A meta-model [36] and its implementation CArtAgO [38].

Since in the agent paradigm each agent is an independent locus of control, coordination means become essential towards regulating the overall behavior of the system. As it is well underlined in [29], the agent-based model allows to naturally tackle the issue of coordination by introducing the concepts of *interaction protocol* [18], and that of *norm* [26, 46]. These concepts are at the heart of the design of multiagent systems. The deliberative cycle of agents is affected by the norms and by the obligations these norms generate as a consequence of the agents' actions. Each agents is free to adapt its behavior to (local or coordination) changing conditions, e.g., by re-ranking its goals based on the context or by adopting new goals.

Institutions and organizations set the ground for coordination and cooperation among agents. Intuitively, an institution is an organizational structure for coordinating the activities of multiple interacting agents, that typically embodies some rules (norms) that govern participation and interaction. In general, an organization adds to this societal dimension a set of organizational goals, and powers to create institutional facts or to modify the norms and obligations of the normative system [8]. Agents, playing one or more roles, must accomplish the organizational goals respecting the norms. Institutions and organizations are, thus, a way to realize functional decomposition in an agent setting.

5 The Rise of the Action Force

Actions are the capabilities agents have to modify their environment. The process force is mapped onto a cycle in which the agent observes the world (updating its beliefs), deliberates which intentions to achieve, plans how to achieve them, and finally executes the plan [12]. Beliefs and intentions are those components of the process abstraction that create a bridge respectively towards the object/data

[5] Summarizing, objects "do it" for free because they are data, agents are processes and "do it" because it is functional to their objectives.

force (i.e., the environment) and the action force. Beliefs concern the environment. Intentions lead to action [49], meaning that if an agent has an intention, then the expectation is that it will make a reasonable attempt to achieve it. In this sense, intentions play a central role in the selectin and the execution of action. Consequently, instead of being subordinate to the process force the action force is put in relation to it by means of intentions. This is a difference with respect to functional decomposition, where actions are produced by refining a given goal through a top-down strategy.

A fundamental step towards raising the value of the action force is brought by *normative multiagent systems* [30, 9], which take inspiration from mechanisms that are typical of human communities, and have been widely studied in the research area on multiagent systems. According to [9] a normative multiagent system is: "a multiagent system together with normative systems in which agents on the one hand can decide whether to follow the explicitly represented norms, and on the other the normative systems specify how and in which extent the agents can modify the norms". Initially the focus was posed mainly on *regulative norms* that, through obligations, permissions, and prohibitions, specify the patterns of actions and interactions agents should adhere to, even though deviations can still occur and have to be properly considered [30]. More recently, regulative norms have been combined with *constitutive norms* [8, 17, 21], which support the creation of institutional realities by defining institutional actions that make sense only within the institutions they belong to. A typical example is that of "raising a hand", which counts as "make a bid" in the context of an auction. Institutional actions allow agents to operate within an institution. Citing [21], the impact on the agent's deliberative cycle is that agents can "reason about the social consequences of their actions". In this light, going back to Meyer's forces, if agents are abstractions for processes and environments for objects, then *norms* are abstractions of the *action force* (see Figure 2) because norms model actions and, thus, condition the way in which processes operate on objects. In fact, norms specify either institutional actions, or the conditions for the use of such actions, consequently regulating the acceptable behavior of the agents in a system. This view is also supported by the fact that norms concern "doing the right thing" rather than "doing what leads to a goal" [46].

6 Data and Norm Awareness

The difficulty of engineering multiagent systems lies in the fact that the environment includes a process but such process is typically not represented in a way that can be reasoned about. Not only the environment should be given in terms of a data information model, specifying the structure of the information, and a data lifecycle, specifying data state transitions, (*data awareness*) but the two should be explicitly represented and accessible to the agents in their deliberative cycle. To this aim, such an explicit representation should constitute a body of *norms*, which describe how data evolution is governed, allowing agents to reason about the consequences of their actions and to have expectations about

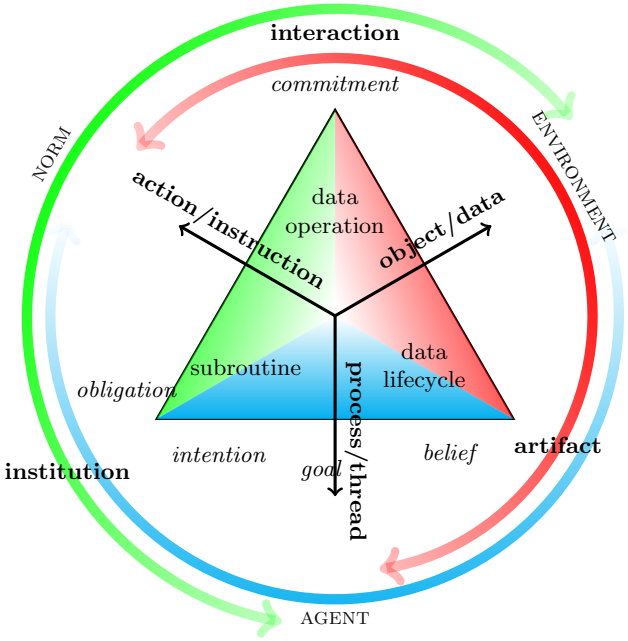


Fig. 2. Rereading Meyer's forces.

the evolution of the environment (*norm awareness*). Only a holistic, data- and norm-aware view would allow agents to reason and to have expectations on the evolution of the *whole* system, autonomously deciding the course of actions to apply.

Such a holistic solution where constitutive norms are used to specify both agent actions and data operations, and where regulative norms are used to create expectations on the overall evolution of the system (agents behavior and environment evolution) is, however, still missing. Object-Orientation associates operations to data; the set of executable operations can sometimes change along time depending on an object's lifecycle, but the paradigm did not push the study towards a normative representation. Similarly, while business artifacts provide both a rich description of their data and their lifecycle, they do not provide any link to a corresponding normative understanding, thus making impossible for the agents to leverage this knowledge for reasoning about how to act. Artifacts in the A&A model are radically different from the business artifacts because they do not come with an explicit information model for data, and they do not expose their lifecycle. Consequently, this lifecycle information cannot be exploited at design time, nor at runtime to reason about which actions should be taken towards the achievement of the agent goals.

A data- and norm-aware perspective would also bring advantages from a software engineering perspective, mainly residing in an increased decoupling among the agent system components, in a way that resembles what happens with business artifacts [20]. This is due to the fact that, at design time, norms would provide a programming interface between agents and their environment, given in terms of those state changes that are relevant in the environment.

6.1 A data-centric Approach to Interaction

A first step in the direction of having data and norm awareness is provided by the JaCaMo+ platform [3], which allows Jason agents [11] to engage commitment-based interactions [42], in turn reified as CArtAgO [37] artifacts. JaCaMo+ artifacts implement the social state of the interaction and provide the roles that are then enacted by the agents. The explicit representation of the social state enables the realization of a data-aware approach, where the data are the events occurring in the social state, while commitments provide the information necessary to agents in their interaction. Both agents and artifacts, encoding social states, are first-class elements in the design of the multiagent system. A commitment $C(x, y, s, u)$ captures that agent x (debtor) commits to agent y (creditor) to bring about the consequent condition u when the antecedent condition s holds. Antecedent and consequent conditions are conjunctions or disjunctions of events and commitments. Besides having an information model commitments have a lifecycle [45] that can be captured by a set of norms [22]. A commitment is *null* right before being created; *active* when it is created. Active has substates: *conditional* (as long as the antecedent condition did not occur), and *detached* (when the antecedent condition occurred, the debtor is engaged in the consequent condition of the commitment). An active commitment can become:

pending if suspended; *satisfied*, if the engagement is accomplished; *expired*, if it will not be necessary to accomplish the consequent condition; *terminated* if the commitment is canceled when conditional or released when active; and finally, *violated* when its antecedent has been satisfied, but its consequent will be forever false, or it is canceled when detached (the debtor will be considered liable for the violation). Commitments in JaCaMo+ belong to the social state and are shared by the interacting agents as resources. So, they are information, that is created and evolves along the interaction with event occurrence, and that contributes to the specification of the environment in which the agents operate. In this light, the social state can be seen as a special kind of business artifact in the sense of [7, 20, 15]. JaCaMo+ allows specifying agent programs as Jason plans, whose triggering events amount to the change of the state of some commitment [2]. Suppose, to make an example, that the commitment goes to the state "detached" and that this event triggers a plan in the agent which is the debtor of that commitment: the connection between the commitment and the associated plan is not only causal (event triggers plan), but rather the plan is explicitly attached to the commitment, in the sense that its aim is to satisfy the consequent condition of the commitment (norm-awareness).

While the representation of commitments in the JaCaMo+ platform is propositional, the Cupid language [19] provides a more sophisticate and information-centric representation that distinguishes between a schema (what occurs in a specification) and its instances (what transpires and is represented in a database), reserving the term commitment only for schemas. This avoids the inadequacy of first-order in representing commitment instances by relying on relational database queries. The advantages, brought to the analysis of properties, of a data-aware approach are proved in DACMAS [34], which incorporates commitment-based MASs but in a data-aware context. In general, in presence of data transition systems become typically infinite-state [15]. On the one hand, this is due to the fact that there is no bound on the number of tuples that can be added to database relations as the computation goes on. On the other hand, even when the number of tuples does not exceed a certain threshold, it is possible to populate them using infinitely many different data objects. Interestingly, when a DACMAS is state-bounded, i.e., the number of data that are simultaneously present at each moment in time is bounded, verification of rich temporal properties becomes decidable. Notably, this shows that, by suitably controlling how data are evolved in the system, it is possible to make agents data-aware without compromising their reasoning capabilities [6, 34].

7 Conclusion

Section 2 introduced properties that characterize quality software. Let us see how the rereading of Meyer's forces, that is depicted in Figure 2, impacts on the desired qualities of software. *Robustness* is the ability to react appropriately to abnormal conditions. The view of the action force as captured by norms allows agents to reason on the lifecycle of data in the environment, thus adding to the

already available capability of reasoning about deviations from agent's expected behavior, an enhanced capability of reasoning about abnormal conditions in the environment and decide how to react to them. So, in principle, the robustness of the system should be increased. The fact that data structure and lifecycles are explicitly represented in a way that can be reasoned about makes agents and their environment more decoupled, avoiding the need of customizing agent programs depending on the environment. This, in turn, increases both *extendibility* and *reusability* of all the components of the MAS. Last but not the least, data-awareness joint with a norm-based representation both enables a fully fledged range of verifications and helps modularizing the verification of properties inside a MAS, thus enhancing the *correctness* quality. In particular, if norms allow both for the specification of the environment and for the specification of action, it becomes possible to perform the analysis of properties at the level of norms rather than on the system as a whole. For instance, given a coordination artifact, it will be possible to verify deadlock freedom on the norms that it encodes and that represent it. The outcome will hold for any instance of the artifact that will be created. Of course, for each use it will be necessary to check that the usage of the artifact, done by a specific agent, conforms to the specification but this is a much simpler kind of verification [4]. A language for representing norms that guarantees a priori the decidability of property analysis would be a great advancement being the tool that agents need to reason and decide which action to take, thus leveraging their autonomy. JaCaMo [10], simpAL [39], JaCaMo+ [2] are existing platforms for the development of MAS that have the right potential for developing the view depicted in Figure 2. The next step would be the introduction of information-centric artifacts, whose lifecycle and data evolution are realized by way of query languages that, as for DACMAS [34], guarantee decidability when certain constraints are met. For commitment-based platforms, the Cupid [19] language would provide analogous features.

Concerning agent-based design, many proposals are found in the literature on Agent-Oriented Software Engineering, where agents are used as high-level software components that are characterized by autonomy and high-level communication, that is based on speech acts. Briefly, SODA [33] is an agent-oriented methodology for the analysis and design of agent-based systems, adopting a layering principle and a tabular representation. It focuses on inter-agent issues, like the engineering of societies and environment for MAS, and relies on a meta-model that includes both agents and artifacts. GAIA [50] is a methodology for developing a MAS as an organization. Tropos [13] is a requirements-driven methodology for developing multiagent systems. The 2CL Methodology [5] is an extension of [24]. It supports the design of commitment-based business protocols that include temporal constraints, and allows the verification of properties. New methodologies, however, are needed to tackle the norm-oriented and data-aware vision that we have illustrated. CoSE [2] is a commitment-driven methodology for programming agents.

Finally, [39] explores agent-oriented programming as a general purpose programming paradigm. It compares agent-based programming to actor-based pro-

gramming from a qualitative perspective, to explain the maturation process that lead to the development of the simpAL programming language. The simpAL languages is grounded on the concepts of agent, artifact, and workspace. A simpAL program is an organization where agents play roles. A static typing mechanism is provided to enact compile-time verifications concerning the implementation and interaction of agents and artifacts. [16] compares Jason, as a representative of agent programming language, against Erlang and Scala, that are two actor-oriented programming languages, in a communication benchmark, in order to verify if actor languages have better performances. The reported quantitative results (which concern time, memory and core usage), show that despite the fact that agent programming languages require a significant overhead when used to develop complex agents, Jason has reasonable performance.

Acknowledgements. The authors would like to thank the anonymous reviewers for the helpful comments. This work was developed during the sabbatical year that Matteo Baldoni and Cristina Baroglio spent at the Free University of Bolzano-Bozen. It was partially supported by the *Accountable Trustworthy Organizations and Systems (AThOS)* project, funded by Università degli Studi di Torino and Compagnia di San Paolo (CSP 2014).

References

1. Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services*. Springer, 2004.
2. Matteo Baldoni, Cristina Baroglio, Federico Capuzzimati, and Roberto Micalizio. Empowering agent coordination with social engagement. In Marco Gavanelli, Evelina Lamma, and Fabrizio Riguzzi, editors, *AI*IA 2015, Advances in Artificial Intelligence - XIVth International Conference of the Italian Association for Artificial Intelligence, Ferrara, Italy, September 23-25, 2015, Proceedings*, volume 9336 of *Lecture Notes in Computer Science*, pages 89–101. Springer, 2015.
3. Matteo Baldoni, Cristina Baroglio, Federico Capuzzimati, and Roberto Micalizio. Leveraging Commitments and Goals in Agent Interaction. In D. Ancona, M. Maratea, and V. Mascardi, editors, *Proc. of XXX Italian Conference on Computational Logic, CILC*, 2015.
4. Matteo Baldoni, Cristina Baroglio, Amit K. Chopra, Nirmit Desai, Viviana Patti, and Munindar P. Singh. Choice, Interoperability, and Conformance in Interaction Protocols and Service Choreographies. In K. Decker, J. Sichman, C. Sierra, and C. Castelfranchi, editors, *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2009*, pages 843–850, Budapest, Hungary, May 2009. IFAAMAS.
5. Matteo Baldoni, Cristina Baroglio, Elisa Marengo, Viviana Patti, and Federico Capuzzimati. Engineering commitment-based business protocols with the 2CL methodology. *JAAMAS*, 28(4):519–557, 2014.
6. Francesco Belardinelli, Alessio Lomuscio, and Fabio Patrizi. A computationally-grounded semantics for artifact-centric systems and abstraction results. In Toby Walsh, editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 738–743. IJCAI/AAAI, 2011.

7. Kamal Bhattacharya, Nathan S. Caswell, Santhosh Kumaran, Anil Nigam, and Frederick Y. Wu. Artifact-centered operational modeling: Lessons from customer engagements. *IBM Systems Journal*, 46(4):703–721, 2007.
8. Guido Boella and Leendert W. N. van der Torre. Regulative and constitutive norms in normative multiagent systems. In Didier Dubois, Christopher A. Welty, and Mary-Anne Williams, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004), Whistler, Canada, June 2-5, 2004*, pages 255–266. AAAI Press, 2004.
9. Guido Boella, Leendert W. N. van der Torre, and Harko Verhagen. Introduction to normative multiagent systems. In Guido Boella, Leendert W. N. van der Torre, and Harko Verhagen, editors, *Normative Multi-agent Systems, 18.03. - 23.03.2007*, volume 07122 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.
10. Olivier Boissier, Rafael H. Bordini, Jomi F. Hübner, Alessandro Ricci, and Andrea Santi. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming*, 78(6):747 – 761, 2013.
11. Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. John Wiley & Sons, 2007.
12. Michael E. Bratman. What is intention? In P. Cohen, J. Morgan, and M. Pollack, editors, *Intensions in Communication*, pages 15–31. MIT Press, Cambridge, MA, 1990.
13. Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.
14. David M. Bridgeland and Ron Zahavi. *Business Modeling: A Practical Guide to Realizing Business Value*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
15. Diego Calvanese, Giuseppe De Giacomo, and Marco Montali. Foundations of data-aware process analysis: a database theory perspective. In Richard Hull and Wenfei Fan, editors, *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, pages 1–12. ACM, 2013.
16. Rafael C. Cardoso, Jomi Fred Hübner, and Rafael H. Bordini. Benchmarking communication in actor- and agent-based languages. In Massimo Cossentino, Amal El Fallah-Seghrouchni, and Michael Winikoff, editors, *Engineering Multi-Agent Systems - First International Workshop, EMAS 2013, St. Paul, MN, USA, May 6-7, 2013, Revised Selected Papers*, volume 8245 of *Lecture Notes in Computer Science*, pages 58–77. Springer, 2013.
17. Amit K. Chopra and Munindar P. Singh. Constitutive interoperability. In *Proceedings of the 7th Int. J. Conf. on Autonomous agents and multiagent systems, Volume 2*, pages 797–804. International Foundation for Autonomous Agents and Multiagent Systems, 2008.
18. Amit K. Chopra and Munindar P. Singh. Agent communication. In Gerhard Weiss, editor, *Multiagent Systems, 2nd edition*. MIT Press, 2013.
19. Amit K. Chopra and Munindar P. Singh. Cupid: Commitments in relational algebra. In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 2052–2059. AAAI Press, 2015.
20. David Cohn and Hull Richard. Business Artifacts: A Data-centric Approach to Modeling Business Operations and Processes. *IEEE Data Eng. Bull.*, 32(3):3–9, 2009.

21. Natalia Criado, Estefania Argente, Pablo Noriega, and Vicent Botti. Reasoning about constitutive norms in bdi agents. *Logic Journal of IGPL*, 2013.
22. Mehdi Dastani, Leendert van der Torre, and Neil Yorke-Smith. Commitments and interaction norms in organisations. *J. Autonomous Agents and Multiagent Systems*, pages 1–43, 2015.
23. Yves Demazeau. From interactions to collective behaviour in agent-based systems. In *Proceedings of the 1st. European Conference on Cognitive Science*, pages 117–132, Saint-Malo, 1995.
24. Nirmit Desai, Amit K. Chopra, and Munindar P. Singh. Amoeba: A methodology for modeling and evolving cross-organizational business processes. *ACM Trans. Softw. Eng. Methodol.*, 19(2), 2009.
25. Antonio Di Leva and Salvatore Femiano. The bp-m* methodology for process analysis in the health sector. *Intelligent Information Management*, 3(2):56–63, 2011.
26. Jack P. Gibbs. Norms: The problem of definition and classification. *American Journal of Sociology*, 70(5):586—594, 1965.
27. Guido Governatori. Law, logic and business processes. In *Third International Workshop on Requirements Engineering and Law, RELAW 2010, Sydney, NSW, Australia, September 28, 2010*, pages 1–10. IEEE, 2010.
28. Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In Nils J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, August 1973*, pages 235–245. William Kaufmann, 1973.
29. Nicholas R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117(2):277–296, 2000.
30. Andrew J.I. Jones and José Carmo. Deontic logic and contrary-to-duties. In Dov Gabbay, editor, *Handbook of Philosophical Logic*, page 203–279. Kluwer, 2001.
31. Bertrand Meyer. *Object-oriented Software Construction (2Nd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.
32. John C. Mitchell. *Concepts in programming languages*. Cambridge University Press, Cambridge, New York (N. Y.), 2002.
33. Ambra Molesini, Andrea Omicini, Enrico Denti, and Alessandro Ricci. SODA: A roadmap to artefacts. In *Engineering Societies in the Agents World VI*, volume 3963 of *LNAI*, pages 49–62. Springer, 2006. 6th Int. Workshop (ESAW 2005).
34. Marco Montali, Diego Calvanese, and Giuseppe De Giacomo. Verification of data-aware commitment-based multiagent system. In Ana L. C. Bazzan, Michael N. Huhns, Alessio Lomuscio, and Paul Scerri, editors, *International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '14, Paris, France, May 5-9, 2014*, pages 157–164. IFAAMAS/ACM, 2014.
35. Rumyana Neykova and Nobuko Yoshida. Multiparty Session Actors. In eva Kühn and Rosario Pugliese, editors, *Coordination Models and Languages - 16th IFIP WG 6.1 International Conference, COORDINATION 2014, Held as Part of the 9th International Federated Conferences on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*, volume 8459 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2014.
36. Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3):432–456, December 2008. Special Issue on Foundations, Advanced Topics and Industrial Perspectives of Multi-Agent Systems.
37. Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Artifacts in the A&A meta-model for multi-agent systems. *JAAMAS*, 17(3):432–456, 2008.

38. Alessandro Ricci, Michele Piunti, and Mirko Viroli. Environment programming in multi-agent systems: an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems*, 23(2):158–192, 2011.
39. Alessandro Ricci and Andrea Santi. From Actors and Concurrent Objects to Agent-Oriented Programming in simpAL. In Gul A. Agha, Atsushi Igarashi, Naoki Kobayashi, Hidehiko Masuhara, Satoshi Matsuoka, Etsuya Shibayama, and Kenjiro Taura, editors, *Concurrent Objects and Beyond - Papers dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday*, volume 8665 of *Lecture Notes in Computer Science*, pages 408–445. Springer, 2014.
40. Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.
41. Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, March 1993.
42. Munindar P. Singh. An ontology for commitments in multiagent systems. *Artif. Intell. Law*, 7(1):97–113, 1999.
43. Munindar P. Singh and Michael N. Huhns. *Service-oriented computing - semantics, processes, agents*. Wiley, 2005.
44. Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. Why Do Scala Developers Mix the Actor Model with Other Concurrency Models? In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 302–326, Berlin, Heidelberg, 2013. Springer-Verlag.
45. Pankaj R. Telang, Munindar P. Singh, and Neil Yorke-Smith. Relating Goal and Commitment Semantics. In *Post-proc. of ProMAS*, volume 7217 of *LNCS*. Springer, 2011.
46. Göran Therborn. Back to norms! on the scope and dynamics of norms and normative action. *Current Sociology*, 50:863–880, 2002.
47. Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer, 2007.
48. Danny Weyns, Andrea Omicini, and James Odell. Environment as a first class abstraction in multiagent systems. *JAAMAS*, 14(1):5–30, 2007.
49. Michael J. Wooldridge. *Introduction to multiagent systems, 2nd edition*. Wiley, 2009.
50. Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems: The Gaia methodology. *ACM Trans. Softw. Eng. Methodol.*, 12(3):317–370, 2003.

Agent-Oriented Methodology for Designing Cognitive Agents for Serious Games

Cheah Wai Shiang[1,*], John-Jules Meyer[*], Kuldar Taveter[2,#]

[1]Faculty of Computer Science and Information Technology, Universiti Malaysia Sarawak, Kota Samarahan, Malaysia
c.waishiang@gmail.com
[*]Department of Informatics, Universiteit Utrecht, Utrecht, The Netherlands
J.J.C.Meyer@uu.nl
[2]Department of Informatics, Tallinn University of Technology, Tallinn, Estonia
[#]Department of Electrical Engineering, University of Shanghai for Science and Technology, P.R. China
kuldar.taveter@ttu.ee

Abstract. One of the challenges in the development of serious games is designing believable virtual characters. Although statistical and machine-learning techniques have been introduced to emulate believable virtual characters in serious games, such techniques are computationally expensive. Alternatively, cognitive architectures like ACT-R have been proposed in robotics for achieving human-like robots. The same approach can be adapted for designing cognitive agents for serious games. However, there is neither methodology nor systematic process available for designing of a cognitive architecture for software agents to be used in serious games. As a result, it is hard to model, design and develop cognitive agents. Also, it is hard to transfer a cognitive design of a believable virtual character to other similar kinds of projects. Prompted by these needs, we propose to combine the AOM and Prometheus methodologies to fill this gap. We demonstrate how the combination of AOM and Prometheus is used to model a multi-agent BDI cognitive architecture at the abstraction layers of conceptual domain modelling and platform-independent design and how the resulting tuned cognitive architecture can be implemented in OOAPL – an object-oriented language for programming BDI agents. With the proposed methodology, we can track and trace cognitive processing within a virtual character. Also, we can reuse and adapt agent models to different scenarios or case studies.

Keywords: Cognitive architecture, cognitive modelling, agent-oriented software engineering, methodology, BDI

1 Introduction

Serious games are designed for purposes beyond pure entertainment. The objective of serious games is to facilitate training and public awareness by using games to address serious matters, such as cybersecurity, homeland safety, and crisis management. Serious gaming allows the users to experience situations that are impossible to create in the real world due to cost, safety, time, and complexity [3]. It has been indicated that the challenges of serious games are to emulate or simulate virtual characters or non-player characters (NPC) that behave in the way human beings act in the real world, interact socially with one another, and compete or cooperate to solve problems [11]. Accordingly, the general research question addressed by this article is how to create believable autonomous cognitive characters?

According to Desai and Szafron [4], believable characters are characterized by human-like features such as the capabilities to learn, to pause between decisions, to make mistakes, and to adjust their strategies in response to their opponents' actions. Work has been done to introduce various AI techniques to mimic believable characters. Resulting from this work, game characters imitate and understand the behaviour of a player and interact with the player through different learning algorithms like statistical analysis, machine learning, and learning algorithms based on neural network reinforcement [4].

Statistical methods and machine learning algorithms are computationally expensive [5]. Alternatively, cognitive architectures have been proposed to model and control believable software agents or robots [7, 13, 14, 16, 17]. Cognitive architecture can be defined as the "refinement of the notions of behaviour and behaviour models, and the development of cognitive insertion machines for manipulating of behavioural models of different levels of abstraction" [14]. A well-known cognitive architecture that has been successfully used for simulating believable virtual characters is "Adaptive Control of Thought-Rational" (ACT-R) [16]. The ACT-R architecture consists of a number of modules, each of which is devoted to processing a particular kind of information [16]. The goal module maintains information relevant to the goals of the agent and the declarative module is responsible for encoding, storing, and retrieving of the knowledge by the agent. The vision, audio, speech and motor modules serve as sensors and actuators for the agent. Finally, the procedural module is responsible for coordination between the modules, which is done in sequence [16, 22].

Different serious games require virtual characters with different cognitive capabilities. To cater for this need, a methodology is required for designing virtual characters as software agents in serious games. This kind of methodology would facilitate modelling, designing and developing of tailor-made cognitive agents. Also, such methodology would support transferring the existing cognitive architectures to other, similar kinds of projects.

To fill the gap in designing autonomous and believable cognitive agents for serious games, we propose to use the combination of Agent Oriented Modelling

(AOM) [10] and Prometheus [9] agent-oriented software engineering methodologies. AOM is an agent-oriented software engineering methodology that was introduced to designing complex socio-technical systems of the "human-agent collectives" type [24]. Although AOM has been used for requirements engineering for agent-based serious games [25], the full potential of AOM in designing serious games has not yet been explored. This leads us to the refined research question: how can the combination of AOM and Prometheus methodologies be used to design believable virtual characters for serious games?

Answering the research question of this paper has been inspired by the multi-agent cognitive architecture proposed for robots in [22]. In the cognitive architecture [22], the "brain" of a robot is modelled and designed as a collection of agents, each of which is responsible for a particular functional area, such as collecting or emitting signals and performing reactive tasks at a lower level, signal processing and performing semi-autonomous tasks at the middle level, and performing cognitive and social tasks at a higher level. Each constituent agent of the cognitive robot architecture proposed in [22] has the PDE (Perception-Deliberation-Execution) architecture and incorporates memory capacity to be able to maintain its internal state. No agent has complete control of the robot. Therefore different agents making up the cognitive architecture of a robot have to combine their objectives.

In this article, we propose similar to [22] cognitive architecture for software agents, where the architecture is made up by agents, each of which has the Belief-Desire-Intention (BDI) agent architecture [26] and is responsible for a different functional area, such as perceiving, holding memory, self-identification, situation assessment, planning, and performing actions. The resulting multi-agent BDI cognitive architecture thus consists of a collection of BDI agents that collaborate to perform cognitive processing. To design different multi-agent BDI architectures, the combination of AOM and Prometheus agent-oriented software engineering methodologies is applied. We also demonstrate in this article how a cognitive multi-agent BDI architecture achieved by applying the combination of the AOM and Prometheus methodologies can be implemented by using OOAPL [21] – an object oriented BDI agent programming language.

This article is structured as follows. The motivational case study is introduced in Section 2. It presents two scenarios of fore extinguishing together with the cognitive multi-agent BDI architecture of the virtual characters of these scenarios. Section 3 outlines the details of the combination of the AOM and Prometheus agent-oriented software engineering methodologies for designing believable virtual characters with the multi-agent BDI architectures. The proposed combination of methodologies covers understanding the problem domain for which virtual characters are to be designed by conceptual domain modelling and designing the multi-agent BDI architecture for characters of the given problem domain by platform-independent design. Section 5 addresses the implementation of the agent models created in Section 4 in an object-oriented agent programming language. Section 6 presents a review on other approached of designing cognitive

agents for serious games. That section also reports on the works performed on the integration of BDI agents into game engines. Finally, the conclusions and perspectives for future work are presented in Section 7.

2 Motivational case study

In this section, a motivational case study on cognitive agents participating in a scenario of fire extinguishing is presented. The scenario is elaborated into two variant scenarios of fire extinguishing. The fire extinguishing occurs in a grid network with two rooms, an open space, and two virtual agents. The elaborated scenarios will be used in Section 4 to validate the combined methodology for designing multi-agent BDI cognitive architectures proposed in this article. The two scenarios are described as follows:

Scenario 1: virtualAgent1 is in the building and virtualAgent2 is in the open space. VirtualAgent1 has no fire extinguishing experience, while virtualAgent2 is well trained to extinguish small fires. Suddenly a fire bursts out in the open space. When the fire is starts, virtualAgent1 located in a closed space is unaware of the fire and continues with her work. VirtualAgent2 being located in the open space will take actions. His first action is to find a fire extinguisher. After that he will take the extinguisher, locate the fire, move towards the fire, and extinguish the fire.

Scenario 2: virtualAgent1 is aware of the fire in the room and cries for help. VirtualAgent2 responses by moving in the direction of the cry for help. He grabs the fire extinguisher and asks virtualAgent1 for the location of the fire. He receives her reply on the location of the fire and moves towards it and extinguishes the fire.

Let us now assume that both virtual agents have been designed according to the multi-agent cognitive architecture. Adopted from cognitive multi-agent architectures in robotics [22, 23], the cognitive architecture of a virtual agent is presented in Figure 1. In this architecture, each agent serves to fulfil some kinds of requirements for cognitive agents, such as situation awareness, planning, reasoning, and learning, as has been stated in [7, 13, 14, 16, 29]. It is up to the agent designer to design each given agent as a normative agent, learning agent, and so on, according to the requirements. The cognitive multi-agent BDI architecture depicted in Figure 1 consists of six agents that have been designed to support the cognitive processing of a believable virtual character. Each agent is assigned with an identifier for ease of recognition. The constituent agents of the architecture shown in Figure 1 are plannerAgent (ID1), actionAgent (ID6), memoryAgent (ID2), situationAssessmentAgent (ID3), perceptionAgent (ID4),

and selfIdentificationAgent (ID4). The plannerAgent (ID1) has the responsibility to plan reactions to incoming perceptions. The situationAssessmentAgent (ID3) has the responsibility to monitor and evaluate the situation surrounding the virtual agent. The perceptionAgent (ID4) has the responsibility to handle incoming perceptions from the environment. The selfIdentificationAgent (ID5) has been designed to maintain the personality of the virtual agent. The memoryAgent (ID2) has the responsibility to manage the appearance and changes of objects in the 3D environment in which the agent is embedded. The actionAgent (ID6) is responsible for managing actions requested by others agents.

The advantage of such multi-agent cognitive architecture is having autonomous constituent agents that control different parts of the agent and cooperate to achieve certain objectives. In addition, the multi-agent cognitive architecture is able to produce scalable, fault-tolerant and flexible behaviours while maintaining the requirements for reusability, parallelism, robustness, and modularity [23].

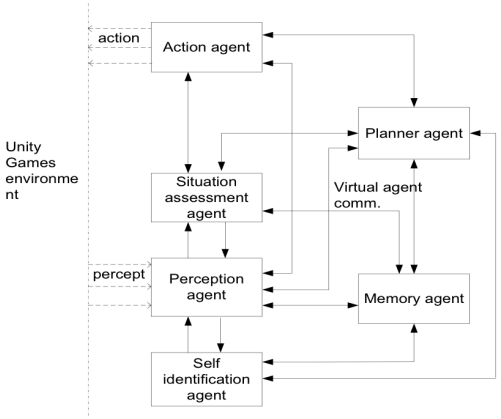


Figure 1. Concrete multi-agent BDI cognitive architecture for a virtual character

3 Methodology for designing cognitive agents

AOM is a comprehensive agent-oriented software engineering methodology that is centred on the viewpoint framework [10]. The framework introduces horizontal abstraction layers and vertical perspectives that are required to focus on when people are involved in engineering open distributed systems. The viewpoint framework has been designed with a reduced number of aspects, as compared, for example, to the Zachman framework [11], to allow people to grasp the aspects more easily. In addition, the viewpoint framework is compliant with the model-driven architecture (MDA) [10]. In AOM this means that the agent models of one abstraction layer can be transformed into the corresponding models at another abstraction layer of AOM.

When designing and implementing a socio-technical system, one is involved in performing a sequence of modelling activities. In brief, the modelling process of AOM covers the abstraction layers of conceptual domain modelling, platform-independent design, and platform-specific design and implementation. The layer of conceptual domain modelling constitutes a high-level motivation layer of the system. It provides a description at the level that allows non-technical stakeholders of any given problem domain to elicit, represent, understand, and discuss the requirements for the system to be designed. The highest layer is not dedicated to any technology to be used for designing the system. The layer of platform-independent design corresponds to the designer view of the system in which the design of the system is decided and represented. However, the design descriptions presented at this layer are not related to any particular implementation platform or language. The design layer instead provides a description that can be turned into a particular implementation at the next layer – the layer of platform-specific design and implementation. The design description at that layer allows the system to be deployed and executed in a particular environment like specific platform, hardware, technology, and architecture.

In more detail, the modelling process at the layer of conceptual domain modelling involves modelling the goals, roles, interactions and domain knowledge. This is followed by deciding agent types, knowledge by agents, interactions between agents and agent behaviours at the layer of platform-independent design. Finally, platform-specific models are created at the layer of platform-specific design and implementation. The modelling process is shown in Figure 2. In addition, each model can be transformed into another model by following the direction of "derives" from arrows depicted in Figure 2. The transformation guidelines are introduced in [10].

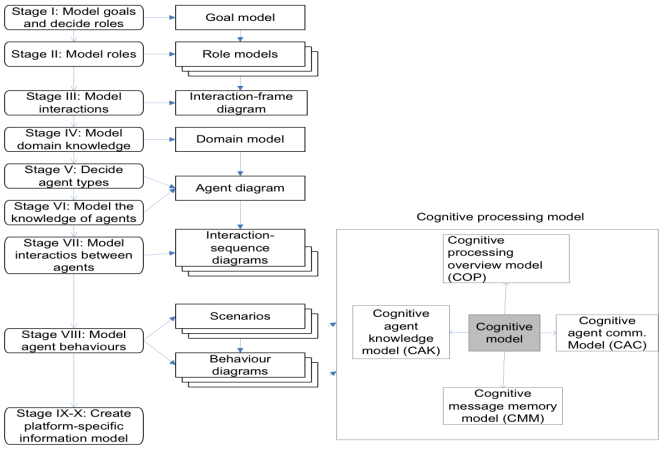


Figure 2. Extension of the AOM methodology by Prometheus for cognitive agents

Figure 2 shows an extension of the stage VIII of the modelling process of AOM by the models from the Prometheus methodology. The extension is needed to cater for concrete design of the multi-agent BDI cognitive architecture as described in section 2. Since AOM addresses abstract agent architecture instead of the BDI architecture, the integration of AOM with Prometheus [9] was required for the research work that is being reported by this article. We therefore claim here like it has been claimed in the article [12] that while AOM supports well requirements' elicitation and analysis of cognitive processing, Prometheus efficiently supports initial design of cognitive agents. Hence, a comprehensive combined methodology is introduced for designing cognitive agents.

In the cognitive processing modelling depicted in Figure 2, we first model details of a scenario in the case study. Thereafter we represent the cognitive processing overview model and cognitive memory-message agent model for the given scenario. Finally, the cognitive internal interaction model and cognitive knowledge model are formed to model the details of the cognitive configuration in relation to the given scenario. Cognitive processing modelling is an iterative process until the design goals are satisfied. The details of the models created in the course of the cognitive processing modelling are the following ones:

Cognitive processing overview model (COP-model): Prometheus system overview model is adopted for this purpose to present an overview of the multi-agent BDI cognitive architecture. This model represents the agent types, interaction protocols, perceptions and actions involved in the cognitive processing. **Cognitive memory-message agent model (CMM-model)**: Prometheus agent overview model is adopted to present the overall message flow and the memory utilization strategy for the given agent during cognitive processing. **Cognitive agent communication model (CAC-model)**: AOM interaction diagram is adopted to present the interactions between the agents involved in the cognitive processing. **Cognitive agent knowledge model (CAK-model)**: AOM behaviour model is adopted to present the agent's beliefs and intentions during the cognitive processing.

A methodology for modelling and designing cognitive agent for serious games has been presented in this section. In order to validate and elaborate the methodology, a walkthrough example of the motivational case study is described in Section 4.

4 Designing the architecture for virtual characters in the fire extinguishing scenario

According to the methodology proposed in Section 3, modelling activities start by conceptual domain modelling. In the conceptual domain modelling, the problem domain at hand is analysed and requirements are elicited and represented for the system to be designed. One of the main types of models created at the stage of conceptual domain modelling is goal model.

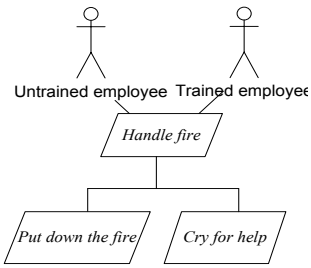


Figure 3. The overall goal model for the scenario of fire extinguishing

Figure 3 presents the overall goal model for the scenario of fire extinguishing. A goal model contains the following components: functional goals, quality goals, and roles. A functional goal represents a functional requirement for the system to be designed. A functional goal can be decomposed into sub-goals, each of which represents a specific aspect of achieving their "parent" goal. Quality goals attached to functional goals represent non-functional requirements. A quality goal sets a specific standard to be targeted when achieving the corresponding functional goal, such as ensuring the customer's satisfaction. Roles attached to functional goals describe the capacities or positions required for achieving the corresponding functional goals. As is illustrated by Figure 3, the main goal of the scenario of fire extinguishing is to "Handle fire". According to our scenario, achieving of this goal involves two roles: Trained Employee and Untrained Employee, both of which are subtypes of the role Employee. The main goal has been elaborated into the two sub-goals: "Extinguish the fire" and "Cry for help". For achieving the main goal along with its two sub-goals, the role Untrained Employee is dependent on the role Trained Employee. Hence, the AOM organization model of the scenario of fire extinguishing depicted in Figure 4 represents a dependency between the roles Untrained Employee and Trained Employee. In general, the organization model of AOM indicates the relationships between the roles attached to the goal model.

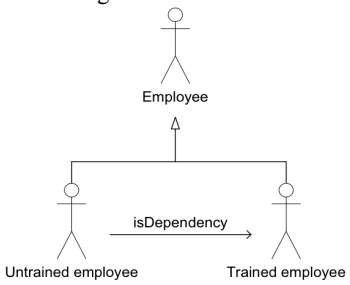


Figure 4. Organization model for the scenario of fire extinguishing

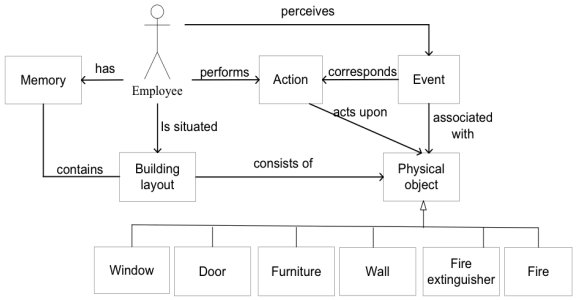


Figure 5. Domain model for the scenario of fire extinguishing

The domain model of the scenario of fire extinguishing is shown in Figure 5. Domain model of AOM captures the knowledge to be represented within the system to be designed. Modelling domain knowledge involves identifying the domain entities and relationships between them. As is illustrated by Figure 5, fourteen domain entities have been modelled for the scenario of fire extinguishing. Agents playing the role of Employee are situated in a building. The "Building layout" consists of "Physical objects" of the types "Wall", "Fire", "Door", "Furniture", "Fire extinguisher", and "Window". All the physical objects are situated in the building and are modelled as contained by the "Memory" domain entity. Agents playing the roles of Trained Employee and Untrained Employee perform actions on physical objects and perceive events associated with physical objects.

At the abstraction layer of platform-independent design, agent types and the shared and private knowledge by agents of these types first have to be decided. In the scenario of fire extinguishing, the role Employee is mapped to the agent type Virtual Agent. There can be several instances of the agent type Virtual Agent. As agents of the type Virtual Agent will be built according to the cognitive multi-agent BDI architecture shown in Figure 1, each instance of Virtual Agent in turn consists of six agents that have been designed to support the cognitive processing of a believable virtual character.

As is shown in Figure 2, one of the central model types in platform-independent design are AOM scenario models. Scenario models of AOM represent for each scenario its goal from the relevant goal model, the initiating agent, the triggering event, and the scenario description consisting of numbered steps of the scenario. Each step models one activity along with the condition for its performing, the involved role and agent type and the involved physical objects. Table 1 represents the high-level scenario for achieving the goal "Put down the fire". According to Table 1, the activity "Act on fire" is elaborated by another scenario – Scenario 2, which is represented as Table 2. The scenario modelled as Table 2 represents the cognitive processing within the multi-agent BDI cognitive architecture for a virtual character represented in Figure 1.

Table 1. The scenario for achieving the goal "Put down the fire"

SCENARIO 1				
Goal	Put down the fire			
Initiator	Virtual Agent			
Trigger	Perceived event associated with the Fire object			
DESCRIPTION				
Condition	Step	Activity	Role / Agent type	Physical objects
	1	Act on fire (**Scenario 1**)	Trained Employee / Virtual Agent	Fire extinguisher Fire

Table 2. The elaborated scenario for achieving the goal "Put down the fire" by a virtual character

SCENARIO 2				
Goal		Put down the fire		
Initiator		situationAssessmentAgent (ID3)		
Trigger		Perceived event associated with the Fire object		
DESCRIPTION				
Cond.	Step	Activity	Agent types and roles involved	Physical objects
	1	Cognition configuration		
	1.1	Subscribe to the Fire object	situationAssessmentAgent (ID3)	Fire
	1.2	Set attentions/state to idle mode	plannerAgent (ID1) actionAgent (ID6)	
	1.3	Subscribe to domain objects	memoryAgent (ID2)	Building layout
	1.4	Subscribe to domain objects	perceptionAgent (ID4)	Building layout
	1.5	Subscribe to body locality	selfIdentificationAgent (ID5)	Locality
	2	Cognition to act on fire		
	2.1	Notify fire	perceptionAgent (ID4) plannerAgent (ID1)	Fire
	2.2	Update attention/state	plannerAgent (ID1)	
	2.3	adoptGoal(act on fire)	plannerAgent (ID1)	
	2.4	Deliberation and execute plan		
	2.4.1	Locate fire extinguisher	plannerAgent (ID1) memoryAgent (ID2)	Fire Fire extinguisher
	2.4.2	Execute traversing plan	plannerAgent (ID1) actionAgent (ID6)	Building layout
Loop	2.4.3	Wait for action status	plannerAgent (ID1) situationAssessmentAgent (ID3)	
	2.4.4	Graps the fire extinguisher	plannerAgent (ID1) actionAgent (ID6)	Fire extinguisher
	2.4.5	Locate fire	plannerAgent (ID1) memoryAgent (ID2)	Fire Fire extinguisher
	2.4.6	Execute traversing plan	plannerAgent (ID1) actionAgent (ID6)	Fire extinguisher
Loop	2.4.7	Wait for action status	plannerAgent (ID1) situationAssessmentAgent (ID3)	Fire Fire extinguisher
	2.4.8	Put down the fire	plannerAgent (ID1) actionAgent (ID6)	Fire Fire extinguisher
	2.5	Update attention/state	plannerAgent (ID1)	

Based on the scenario models of AOM, we will proceed with designing the cognitive capabilities of the involved agents by using models put forward by the Prometheus methodology. Figure 6 presents a cognitive processing overview model for cognitive agents, modelled as a system overview diagram of Prometheus. The diagram represents the five agents involved in cognitive processing – perception agent, action agent, memory agent, planner agent, and self-identification agent – that together simulate a human mind of a virtual character.

Different agents interact by means of the following protocols: "action protocol", "perception protocol" and two domain-specific protocols. The action protocol consists of simple rules that enable receiving action requests from the planner agent and executing these actions. Actions are executed by sending "execute action" commands to the body of the virtual agent.

The perception agent receives from the environment incoming perceptions of time, events, physical objects, and incoming communication and actions by other agents. The perception agent makes the information perceived by it available to other agents by means of the perception protocol. The perception protocol is a publish-subscribe protocol handling incoming perceptions and publishing the perceptions to the agents who have subscribed to perceptions of one or another kind.

As is modelled explicitly in Figure 6, the action agent and situation assessment agent are notified about the new actions or events perceived by means of the "action status" and "event status" messages. In the same way, the memory agent is notified about the physical objects perceived by the "object percept" messages. The memory agent, planner agent, and self-identification agent also use in their work the information contained in the "memory" shared knowledge base.

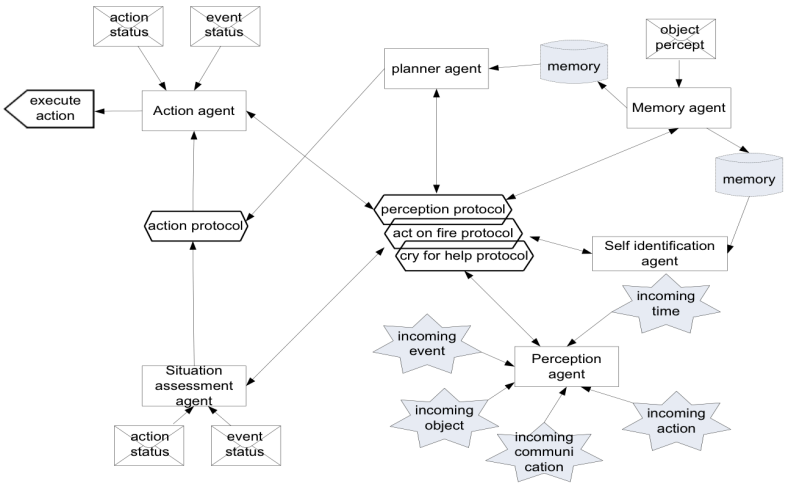


Figure 6. System overview diagram for a cognitive agent

Figure 7 represents a Prometheus agent overview diagram for one of the agents involved in the cognitive architecture – the planner agent. An agent overview diagram models the capabilities of an agent, the triggers of the capabilities and the execution of the capabilities. The planner agent has the following six capabilities: "ask for help", "offer for help", "update agent state", "act on fire", "grasp things", and "use things". Each capability modelled in Figure 7 receives inputs as messages or from the agent memory. The same model also represents strategies that support certain complex capabilities. For example, the "ask for help strategy" supports the "ask for help" capability, and the "explore strategy" and "traversing strategy" support the "act on fire" capability and finally the "offer help strategy" and "traversing strategy" support the "offer help" capability. Meanwhile, strategies also generate messages that request the execution of certain actions. In addition, a strategy may lead to another strategy or capability. For example, the "explore strategy" leads to the "traversing strategy" as well as to the "grasp things" and "use things" capabilities.

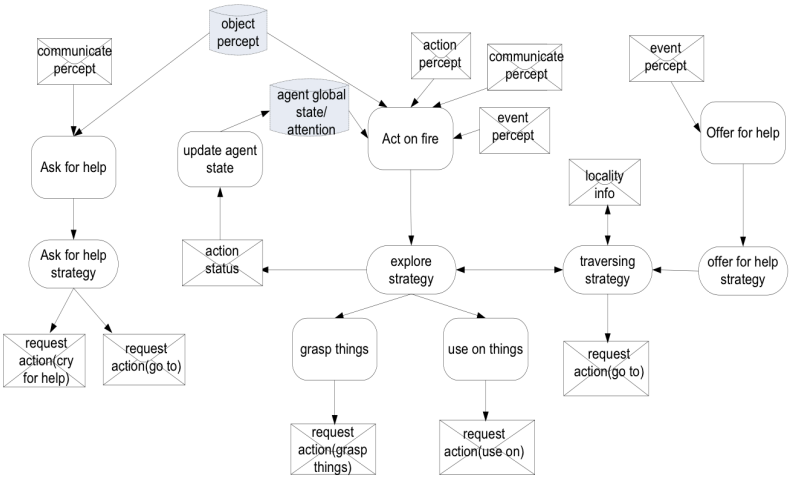


Figure 7. Agent overview diagram for the planner agent

5 Implementation of the fire extinguishing scenario

The previous section explained the platform-independent models for the scenario of fire extinguishing. In this section we focus on the platform-specific design and implementation of the scenario in the object-oriented agent programming language (OOAPL) [21] based on the platform-independent models. OOAPL is a Java-based language with a flexible control and scalability of the agent deliberation lifecycle for programming BDI agents. In brief, the agent deliberation

lifecycle in OOAPL is parallel, concurrent and distributed. This guarantees a balance between slow and fast deliberation processes. The readers are referred to [21] for a better understanding of the lightweight, scalable and flexible OOAPL agent programming platform. While the mind of a virtual character is implemented as an OOAPL agent, the body of the agent is implemented by the CIGA middleware. The CIGA middleware [6] supports the development of non-player characters in virtual worlds.

Figure 8 shows two screenshots on putting out the fire by virtual characters. The agents are situated in a "grid world" consisting of two rooms and an open space. The screenshot on the left shows a virtual agent with an intention to put out the fire upon perceiving the fire. The screenshot on the right represents the interactions between two virtual agents while dealing with the fire.

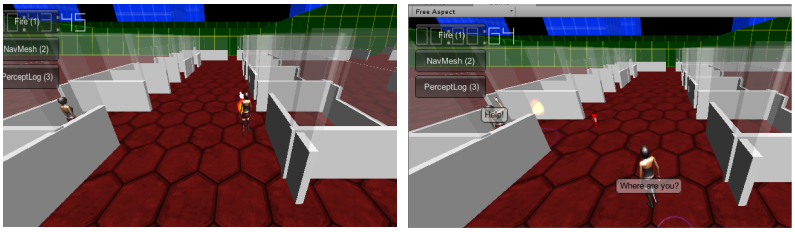


Figure 8. Virtual agents dealing with the fire

Platform-specific design and implementation of the virtual characters shown in Figure 8 is straightforward. A software agent implementing the virtual character subscribes to physical objects of the Fire type, and upon getting notified about a fire object triggers the goal for its subsequent activities. At the beginning of executing the scenario, we set one of the virtual agents to the idle mode. The virtual agent with this kind of behaviour does nothing except observing its environment and paying attention to a Fire object that has occurred in its surrounding "grid world". Once the virtual agent has perceived a fire, the agent locates the fire extinguisher, moves close to the fire extinguisher, grasps the fire extinguisher, locates the fire, moves the extinguisher closer to the fire and uses the fire extinguisher to extinguish the fire. The interaction between the agents making up the architecture of the virtual agent is based on the "act of fire protocol" modelled in Figure 8.

A screenshot shown in the right of Figure 8 shows a variant of the scenario with two virtual agents located in the "grid world". The agents are located in two different places. The first agent is situated in an open space, while the second agent is situated in a room. The first agent has been trained to extinguish a fire, whereas the second agent does not know how to put out the fire. Once a fire occurs in the room, the second agent will therefore only cry for help. As a reaction to the cry by the second agent, the first agent starts looking for the first agent. Thereafter the first agent asks for the location of the second agent. After that the first agent locates the fire extinguisher, moves close to the fire extinguisher, grasps the fire extinguisher, moves towards the location of the second agent, locates the fire and uses the fire extinguisher to extinguish the fire.

6 Related work

Agent-oriented software engineering methodologies for serious games have not been much addressed in the literature. Most of the research work in this area addresses the integration of agent technology with game engines.

The notion of social intelligence has been introduced to regular BDI virtual agents as is described in [12]. The article [12] describes applying virtual BDI agents with cultural parameters for intercultural role-playing simulation games. A goal-based BDI agent for game bots is introduced in [19]. In that case, the GOAL programming language has been used to model and implement the virtual agent. Work has been done to integrate BDI programming platforms like AgentSpeak, GOAL [19], Jason [18], 2APL [6], JACK, and Jadex[20] into game engines like Open Wanderland [1], Unity [6, 16], and Unreal engine [19]. In general, that work is focused on techniques how to integrate BDI programming platforms into a game engines. The article [16] describes how the ACT-R agent architecture has been integrated with the Unity engine for controlling the behaviour of a virtual robot in the Unity-based virtual environment.

7 Conclusions

Cognitive processing within a believable virtual character is complex. This complexity can be tamed by using multi-agent technology in building such virtual characters, which results in the multi-agent BDI architecture proposed in this article. Different agents making up this kind of architecture of virtual characters can be fully reactive, fully deliberative, partially reactive or partially deliberative. This paper presents the combination of the AOM and Prometheus methodologies for modelling the cognitive capabilities of agents by tuning their multi-agent BDI cognitive architectures. Extension of AOM by Prometheus is required because AOM does not support the design of BDI agents. By means of the combined methodology, cognition by a software agent for a virtual character is modelled at the abstraction layers of conceptual domain modelling, platform-independent design, and platform-specific design and implementation. In summary, the proposed in this article combined methodology supports the conceptualization of cognitive agents that is closer to the concerns of a problem domain at hand and is easier to understand and validate, and also improves the efficiency and quality of the cognitive agent development process. Furthermore, the proposed methodology reduces the complexity of developing cognitive agents. In the future, more empirical studies are required to further justify the benefits of the combination of AOM and Prometheus in designing cognitive agents for serious games.

Acknowledgement. The third author of this article has been funded by Shanghai Foreign Talent Scholarship from P.R. China for doing the research work reported in the article.

References

[1] McClure, G., Chang, M., & Lin, F. (2013, December). MAS controlled NPCs in 3D virtual learning environment. In *Signal-Image Technology & Internet-Based Systems (SITIS), 2013 International Conference on* (pp. 1026-1033). IEEE.

[2] Gentile, M., La Guardia, D., Dal Grande, V., Ottaviano, S., & Allegra, M. (2014). An Agent Based Approach to designing Serious Game: the PNPV case study. *International Journal of Serious Games*, *2*(1).

[3] Susi, T., Johannesson, M., & Backlund, P. (2007). Serious games: An overview. *University of Skövde Technical Report HS-IKI-TR-07-001*. Skövde, Sweden.

[4] Desai, N., & Szafron, D. (2012, October). Enhancing the Believability of Character Behaviors Using Non-Verbal Cues. In: *Artificial Intelligence and Interactive Digital Entertainment Conference* (AIIDE).

[5] Tence F., Buche C., Loor P. D., & Marc O. (2010). The challenge of believability in video games: Definitions, agents models and imitation learning. *Proceedings of the 2nd Asian Conference on Simulation and AI in Computer Games, 38–45. Eurosis. 7*(50).

[6] Van Oijen, J., Van Doesburg, W., & Dignum, F. (2011). Goal-based communication using BDI agents as virtual humans in training: An ontology driven dialogue system. In: *F. Dignum (Ed.), Agents for Games and Simulations II, 38-52*. Berlin, Germany: Springer.

[7] Duch, W., Oentaryo, R. J., & Pasquier, M. (2008, June). Cognitive Architectures: Where do we go from here? In: *Proceedings of the Second Conference on AGI, Vol. 171*, 122-136.

[8] Hindriks, K. V., Van Riemsdijk, B., Behrens, T., Korstanje, R., Kraayenbrink, N., Pasman, W., & De Rijk, L. (2011). Unreal goal bots. In: F. Dignum (Ed.), *Agents for games and simulations II* (pp. 1-18). Berlin, Germany: Springer.

[9] Padgham, L., & Winikoff, M. (2002). Prometheus: A methodology for developing intelligent agents. In *Agent-oriented software engineering III* (pp. 174-185). Berlin, Germany: Springer.

[10] Sterling, L., & Taveter, K. (2009). *The Art of Agent-Oriented Modeling*. Cambridge, MA, and London, England: MIT Press.

[11] Zachman, J.A. (1987). A framework for information systems architecture. *IBM Systems Journal, 26* (3)

[12] Luo, Y., Sterling, L., & Taveter, K. (2007, October). Modelling a smart music player with a hybrid agent-oriented methodology. In *Requirements Engineering Conference, 2007. RE'07. 15th IEEE International* (pp. 281-286). IEEE.

[13] Cooper, R., Fox, J., Farringdon, J. & Shallice, T. (1996). A Systematic Methodology for Cognitive Modeling. *Artificial Intelligence, 85* (3-44) (1996)

[14] Letichevsky, A. (2014). Theory of Interaction, Insertion Modeling and Cognitive Architectures. *Biologically Inspired Cognitive Architectures, 8* (19-32).

[15] Gascueña, J. M., & Fernández-Caballero, A. (2009). Agent-based modeling of a mobile robot to detect and follow humans. In: Håkansson, A., Nguyen, N. T., Hartung, R. L., Howlett, R. J. & Jain, L. C. (Eds.) *Agent and Multi-Agent Systems: Technologies and Applications. LNCS, Vol. 5559* (80–89). Berlin, Germany: Springer.

[16] Smart, P. R., Scutt, T., Sycara, K., & Shadbolt, N. R. (2014). Integrating ACT-R Cognitive Models with the Unity Game Engine. In J. O. Turner, M. Nixon, U. Bernardet, & S. DiPaola (Eds.), *Integrating Cognitive Architectures into Virtual Character Design*. IGI Global.

[17] Fox. J. (2000). Making a mind: A cognitive eengineering approach. In: *Conference on How to design a functional mind*. AISB Convention.

[18] Sioutis, C., Ichalkaranje, N., & Jain, L. C. (2003, December). A Framework for Interfacing BDI agents to a Real-time Simulated Environment. In: *Proceedings of the 3rd International Conference of Hybrid Intelligent Systems* (HIS, pp. 743-748). Amsterdam, The Netherlands: IOS Press.

[19] Hindriks, K. V., Van Riemsdijk, B., Behrens, T., Korstanje, R., Kraayenbrink, N., Pasman, W., & De Rijk, L. (2011). Unreal goal bots. In: F. Dignum (Ed.), *Proceedings of the Second Workshop on Agents for Games and Simulations* (pp. 1-18). Berlin, Germany: Springer.

[20] Korecko, S., Sobota, B., & Curilla, P. (2014, November). Emotional agents as non-playable characters in games: Experience with Jadex and JBdiEmo. In: *Computational Intelligence and Informatics (CINTI), 2014 IEEE 15th International Symposium on* (pp. 471-476). IEEE.

[21] Dastani, M., & Testerink, B. (2014). From Multi-Agent Programming to Object Oriented Design Patterns. In *Engineering Multi-Agent Systems* (pp. 204-226). Berlin, Germany: Springer.

[22] Jose, V. B., & Francisco, M. (2011). Robotic control systems based on bioinspired multi-agent systems. *International Journal of Advanced Engineering Sciences and Technologies, 8* (1), 32-38.

[23] Xie, W., Ma, J., Yang, M., & Zhang, Q. (2012). Research on classification of intelligent robotic architecture. *Journal of Computers, 7* (2), 450-457.

[24] Jennings, N. R., Moreau, L., Nicholson, D., Ramchurn, S., Roberts, S., Rodden, T., and Rogers, A. (2014). Human-Agent Collectives. *Communications of the ACM, 57* (12).

[25] Havlik, D.; Deri, O.; Rannat, K.; Warum, M.; Rafalowski, C.; Taveter, K.; Kutschera, P.; Meriste, M. (2015). Training Support for Crisis Managers with Elements of Serious Gaming. In: *Denzer, R., Argent, R.M., Schimak, G., Hřebíček, J. (Ed.). Environmental Software Systems. Infrastructures, Services and Applications* (217−225). Berlin, Germany; Springer.

[26] Rao, A. S., & Georgeff, M. P. (1995, June). BDI agents: From theory to practice. In *ICMAS* (Vol. 95, pp. 312-319).

Augmenting Agent Computational Environments with Quantitative Reasoning Modules and Customizable Bridge Rules

Stefania Costantini[1] and Andrea Formisano[2]

[1] DISIM, Università di L'Aquila
[2] DMI, Università di Perugia, GNCS-INdAM

Abstract. There are many examples where large amount of data might be potentially accessible to an agent, but the agent is constrained by the available budget since access to knowledge bases is subject to fees. There are also several activities that an agent might perform on the web where one or more stages imply the payment of fees: for instance, buying resources in a cloud computing context where the agent's objective is to obtain the best possible configuration of a certain application withing given budget constraints. In this paper we consider the software-engineering problem of how to practically empower agents with the capability to perform such kind of reasoning in a uniform and principled way. To this aim, we enhance the ACE component-based agent architecture by means of a device for practical and computationally affordable quantitative reasoning, whose results actually determine one or more courses of agent's action, also according to policies/preferences.

1 Introduction

There are many examples where large amount of data might be potentially accessible to an agent, but the agent is constrained by the available budget since access to knowledge bases is subject to fees. There are also several activities that an agent may perform on the web on behalf of an user where one or more stages imply the payment of fees. An important example is that of buying resources in a cloud-computing context, where the agent's objective is to obtain the best possible configuration for performing certain tasks in the sense of maximizing performance and minimizing costs, that can anyway stay withing given budget constraints. The work [33] identifies the problem that an agent faces when it has limited budget and costly queries to perform. In order to model such situations, the authors propose a special resource-aware modal logic so as to be able to represent and reason about what is possible to do with a certain budget available. The logic can be adapted to reason separately about cost and time limitation, though an integration is envisaged. Interesting as it is, this work constitutes a good starting point but it presents two problems: (i) such kind of modal logic is computationally hard (though this aspect is not discussed in the aforementioned paper) and thus it can hardly constitute the basis for practical tools; (ii) the axiomatic system of [33] allows one to prove that something can or cannot be achieved within a certain cost. However, an agent needs, in general, to become aware of how goals might possibly be achieved, and should be enabled to choose the best course of action according to its own policies/preferences.

In this paper we tackle some issues related to this problem. First, we consider the software-engineering problem of how to practically empower agents with the capability to perform such kind of reasoning in a uniform and principled way. Second, we consider the adoption of a reasoning device that enables an agent, which may have several costly objectives, to establish which are the alternative possibilities within the available budget, and to select, based upon its preferences, the goals to achieve and the resources to spend, and finally to implement its choice.

Concerning the first aspect, we enhance the Agent Computational Environment (ACE) framework [13], which is a software engineering methodology for designing intelligent logical agents in a modular way. Therefore, in this paper we refer to agent-oriented languages and frameworks which are rooted in Computational Logic. Modules composing an agent interact, in ACE, via *bridge rules* in the style of the Multi-Context Systems (MCS) approach [7, 8, 10]. Such rules take the form of conjunctive queries where each conjunct constitutes a sub-query which is posed to a specific module. Thus, the result is obtained by combining partial results obtained from different sources. The enhancements that we propose here for ACE are based upon the flexible agent-tailored modalities for bridge rules application and for knowledge elaboration defined for the DACMACS framework (Data-Aware Commitment-based managed Multi-Agent-Context Systems), which is aimed at designing data-aware multi-agent-context systems [14, 15]. There, bridge rules are proactively triggered upon specific conditions and the obtained knowledge is reactively elaborated via a *management function* which generalizes the analogous MCS concept.

Second, we extend ACEs so as to include modules for specialized forms of reasoning, including quantitative reasoning. For this kind of reasoning we suggest to adopt the RASP framework [17, 19, 16], which is based upon Answer Set Programming (ASP) and hence it is computationally affordable and reasonably efficient. We show the suitability of such approach by discussing a case study, that will constitute the leading example throughout the paper.

A strong innovation that this paper proposes is that, after obtaining from a reasoning module the description of possible courses of actions, bridge rules "patterns" can be specialized and activated so as to put them into action. This feature is made possible by an enhanced flexible ACE semantics.

The resulting framework can be seen as a creative blend of existing technologies, with some relevant formal and practical extensions. Partially specified bridge rules and their dynamic customization and activation is an absolute novelty and constitutes a relevant advance over MCSs versions, applications and extensions: in fact, bridge rules have been so far conceived as predefined, ground and not amenable to any adaptation. Beyond quantitative reasoning, such more general bridge rules may constitute a powerful flexible device in many applications.

The paper is organized as follows. Section 2 presents a case study that will constitute the leading example throughout the paper. In Section 3 we discuss the quantitative reasoning device we suggest to exploit. Sections 4 and 5 present the enhanced ACE framework and illustrate, on the case study, the dynamic customization of bridge rules. Section 6 introduces the extended ACE semantics and for completeness we provide in Section 7 an actual RASP formalization. Concluding remarks are given in Section 8.

2 Specification of the Case Study

In this section we provide the specification of a case study which we will adopt in the rest of the paper for the illustration of the proposed enhancements to the ACE framework. In Section 7 we will present a realistic implementation in a specific existing approach for quantitative reasoning, shortly introduced in the next section.

We consider a student, that will be represented by an agent which can be seen as her "personal assistant agent". Upon completing the secondary school, she wishes to apply for enrollment to an US university. Each application has a cost, and the tuition fee will have to be paid in case of admission and enrollment. The student has an allotted maximum budget for both. Thus the agent, on behalf of the student, has to reason about: (i) the universities to which an application will be sent; (ii) the university where to enroll, in case a choice can be made.

Actually, the proposed case study is seen as a prototype of a wide number of situations where two kinds of quantitative reasoning are required:

1. The cost of knowledge, as in practical terms a student applies in order to know whether she is admitted.
2. Reasoning under budget limits, as a student may send an application only if: (i) she can afford the fees related to the application; (ii) in case of admission, she can then afford the tuition fees.

If a solution is found considering her preferences and her budget, she will then be able to apply and, if admitted, to enroll. In case more than an option is available, a choice is required so as to select the "best" one according to some criteria.

Without any pretension to precision, we consider the steps that a student has to undergo in order to apply for admission:

1. Pass the general SAT test.
2. Pass the specific SAT test for the subject of interest (such as Literature, Mathematics, Chemistry, etc.)
3. In case of foreign students, pass the TOEFL test.
4. Fill the general application on the application website (that we call collegeorg).
5. Send the SAT results to the universities of interest.
6. Complete the application for the universities of interest.

All these steps are subject to the payment of fees, which are fixed (the fee is independent of the university) for steps 1-4 and depend upon the selected university for steps 5-6. In the example we assume that the student has a budget for the application (say 1500 US dollars) and a limit about the tuition fee she is able to pay (say 22000 US dollars per year). However, she has a list of preferred universities, and within such list she would apply only to universities whose ranking is higher than a threshold. Additionally, since she likes basketball, all other things being equal (*ceteris paribus*) she would prefer universities with the best rankings of the basketball team.

3 Resource-based Reasoning

In the case study, the student's personal assistant agent needs the support of some kind of quantitative reasoning module. Such module should in general be able to provide

the agent, given one or more objectives, with a description of the different ways of achieving the objectives while staying within a budget. A desirable property of the reasoner would be that of allowing preferences and constraints to be expressed about objectives to achieve and modalities for achieving them. A mandatory requisite is the ability to perform such reasoning in a computationally affordable way.

In knowledge representation and reasoning, forms of quantitative reasoning are possible, for example, in Linear Logics and Description Logics. For Linear Logic in particular, several programming languages and theorem provers based on its principles exist (cf. [16] for a discussion). In this paper we adopt RASP (Resource-based ASP) [17, 19], which has in fact been proven in [18] to be equivalent to an interesting fragment of Linear Logic, specifically, to an empowered Horn fragment allowing for a default negation that Linear Logic does not provide (though still remaining within an NP-complete framework). RASP extends ASP, which is a well-known logic programming paradigm where a program may have several "models", called "answer sets", each one representing a possible interpretation of the situation described by the program (cf., among many, [29]). In particular, RASP explicitly introduces in ASP the notion of *resource*, and supports both formalization and quantitative reasoning on consumption and production of resources. RASP also provides complex preferences about spending resources (and in this it is different from the several approaches to preferences that have been defined for ASP, see e.g., [2, 6, 11, 25] and the references therein). Compared with the "competitors", RASP represents possible different uses of a resource and non-determinism in general by means of different answer sets, rather than exploring the various possibilities via backtracking in a Prolog-like fashion. The RASP inference engine is based upon publicly available ASP solvers [35] that are remarkably well-performing and subject of intensive research and development. After the seminal work of [34] one can mention [26, 28, 32, 1, 31, 22], among the most recent developments. Specifically, RASP execution is based upon a front-end module called *Raspberry* which translates RASP programs (via a non-trivial process, see [19] for the details) into ASP. The resulting program can be executed by common ASP solvers.

As a side note, we observe that the clasp ASP solver allows one to add external functions to ASP programs. This is done by defining deterministic functions in a scripting language such as lua or python. Relying on this possibility, one might envisage a re-implementation of the RASP framework exploiting such feature of this specific ASP solver, instead of performing a translation from RASP into ASP, as done in Raspberry. Another recently proposed extension of ASP is H-ASP [12], where propositional reasoning is combined with external sources of numerical computation. The main aim of H-ASP is to allow users to reason about a dynamical system by simulating its possible evolutions along a discretized timeline. The external computations are used to compute the system transitions and may involve both continuous and discrete numerical variables. The expressive power of the resulting framework directly depends on the kind of numerical tasks one integrates, and the computational complexity can exceed NP. Clearly, thanks to the generality of ACE, one could integrate modules based on H-ASP in the ACE framework, similarly to what done for RASP. However, in the case of RASP we stay within NP and directly rely on common "pure-ASP" engines without the need of integrating (and encoding) further computational services.

We are not aware of other reasoning frameworks that combine logic and quantitative techniques, apart from the one proposed in [33], which however is not implemented and, as mentioned, in its present form can hardly admit a computationally affordable version. So, there is nowadays no competitor approach to RASP in practical logic-based quantitative reasoning and its applications in the agent's field.

4 Enhancing the ACE Framework

The ACE framework as defined in [13] considers an agent as composed of:
1) the "main" agent program;
2) a number of Event-Action modules for Complex Event Processing;
3) a number of external contexts the agent can access in order to gather information.

ACE is therefore a highly modular architecture, where the composing modules communicate via *bridge rules* (to be seen below) in the style of Multi-Context Systems (MCSs) [7, 8, 10]. MCSs constitute in fact a particularly interesting approach for modeling information exchange among heterogeneous sources because, within a neat formal definition, it is able to accommodate real heterogeneity of sources by explicitly representing their different representation languages and semantics. The same holds for ACEs, where: external contexts are understood as in MCS, i.e., they can be queried but cannot be accessed in any other way; and where the "local" agent's modules (main agent program and event-action modules) can be defined in any agent-oriented computational-logic-based programming language, such as, e.g., DALI, AgentSpeak, GOAL, 3APL, METATEM, KGP, etc. (see [3, 4, 5, 20, 21, 24, 27, 30] and the references therein), or also in other logic formalisms such as, e.g., ASP (see [29] and the references therein).

In the present setting, we augment the framework with a set of *Reasoning Modules*, say $R_1, \ldots, R_q$, $q \geq 0$, that we see as specialized modules which are able to perform specific forms of reasoning by means of the best suitable formalism/technique/device. Among such modules we may have quantitative reasoning modules. Therefore, an (enhanced) Agent Computational Environment (ACE) $\mathcal{A}$ is now defined as a tuple

$$\langle A, M_1, \ldots, M_r, C_1, \ldots, C_s, R_1, \ldots, R_q \rangle$$

where module A is the "basic agent", i.e., an agent program written in any agent-oriented language. The "overall" agent is obtained by equipping the basic agent with the following facilities. The M_is are "Event-Action modules", which are special modules aimed at Complex Event Processing, that allow the agent to flexibly interact with a complex changing environment. The R_js are "Reasoning modules", which are specialized in specific reasoning tasks. The C_ks are contexts in the sense of MCSs, i.e., external data/knowledge sources that the agent is able to query about some subject, but upon which it has no further knowledge and no control: this means that the agent is aware of the "role" of contexts in the sense of the kind of knowledge they are able to provide, but is unable in general to provide a description of their behavior/contents or to affect/modify them in any way.

Interaction among ACE's components occurs via *bridge rules*, inspired by those in MCS. They can be seen as Datalog-like queries where however each sub-query can be posed to a different module. In MCS, bridge rules have, in general, the following form:

$$s \leftarrow (c_1 : p_1), \ldots, (c_j : p_j), not\, (c_{j+1} : p_{j+1}), \ldots, not\, (c_m : p_m).$$

The meaning is that the rule is *applicable* and s can thus be added to the consequences of a module's knowledge base whenever each atom p_r, $r \leq j$, belongs to the consequences of module c_r (that can be a context or an event-action module, or the basic agent), while instead each atom p_w, $j < w \leq m$, does not belong to the consequences of c_w. Practical run-time bridge-rule applicability will consist in posing query p_i to context c_i. In case for some of the p_is the context is omitted, then the agent is querying its own knowledge base. The part $(c_1 : p_1), \ldots, (c_j : p_j)$ is the *positive body* of the rule, while the remaining part is the *negative body*.

We introduce the following restriction on bridge rules bodies: the basic agent A can query any other module (and, clearly, if it is situated in a MAS context it can communicate with other agents according to some kind of protocol). The M_is and the R_is can query external contexts and the basic agent. Contexts can only query other contexts, i.e., they cannot access agent's knowledge. We also assume (for simplicity and without loss of generality) that bridge-rule heads are unique, i.e., there are never two bridge rules with the same head.

In Managed MCSs the conclusion s, which represents the "bare" result of the application of the bridge rule, becomes $o(s)$ where o is a special operator, whose semantics is provided by a module-specific *management function*. The meaning is that the result computed by a bridge rule is not blindly incorporated into the "target" module knowledge base. Rather, it is filtered, adapted, modified and elaborated by an operator that can possibly perform any elaboration, e.g. evaluation, format conversion, belief revision. To the extreme, the new knowledge item can even be discarded if not deemed to be useful.

In the basic agent we adopt, with suitable adaptations the special agent-oriented modalities introduced in DACMACS. There, bridge-rule activation and management-function application has been adapted to the specific nature of agent systems. First, while bridge rules in MCSs are conceived to be applied whenever applicable (they can be seen, therefore, as a reactive device), DACMACS provides a proactive application upon specific conditions. Second, the incorporation of bridge rule results via the management function is separated from bridge-rule application. In particular, bridge-rule application is determined by a *trigger rule* of the form

$$Q \textbf{ enables } A(\hat{x})$$

where: Q is a query to agent's internal knowledge-base and $A(\hat{x})$ is the conclusion of one of agent's bridge rules. If query Q (the "trigger") evaluates to true, then the bridge rule is allowed to be applied. A trigger rule is proactive in the sense that the application of a bridge rule is enabled only if and when the agent during its operation concludes Q. The bridge rule will be actually applied according to agent's internal control modalities, and will return its results in $\hat{x}$. The result(s) $\hat{x}$ returned by a bridge rule with head $A(\hat{x})$ will then be exploited via a *bridge-update rule* of the following form (where $\beta(\hat{x})$ specifies the operator, management function and actions to be applied to $\hat{x}$):

$$\textbf{upon } A(\hat{x}) \textbf{ then } \beta(\hat{x})$$

We propose a relevant improvement concerning bridge rules. In particular, in MCSs bridge rules are by definition ground, i.e., they do not contain variables: in [9], it is literally stated that [in their examples] they "*use for readability and succinctness schematic*

bridge rules with variables (upper case letters and '_' [the 'anonymous' variable]) which range over associated sets of constants; they stand for all respective instances (obtainable by value substitution)" where however such "placeholder" variables occur only in the p_is while instead the c_is (contexts' names) are constants. This is a serious expressive limitation, that we have tackled in related work. In fact, we admit variables in both the p_is in bridge-rule bodies and in the head s, to be instantiated at run-time by the queried contexts. We also admit contexts in the body to be selected from a directory according to their *role*. Here, we propose a further relevant enhancement: we allow contexts occurring in the body of main agent's A bridge rules to be instantiated via results returned by ACE's other modules. So, such bridge rules will have the following form:

$$s \leftarrow (\mathcal{C}_1 : p_1), \ldots, (\mathcal{C}_j : p_j), not\,(\mathcal{C}_{j+1} : p_{j+1}), \ldots, not\,(\mathcal{C}_m : p_m).$$

where each $\mathcal{C}_i$ can be either a plain constant (as before) or an expression of the form $m_i(k_i)$ that we call *context designator*, which is a term where m_i can be seen as a(n arbitrary) meta-function indicating the required instantiation, and k_i is a constant that can be seen as analogous to a Skolem constant. Such term indicates the kind of context to which it must be substituted before bridge-rule execution, so it might be, for instance, $university(u)$, $student\_data(sd)$, $treatment\_database(d)$, $diagnostic\_expert\_system(de)$. There is no fixed format, rather it is intended as a designation of the required-for knowledge source, that can be either a knowledge repository or a reasoning module.

A bridge rule including context designators will be indicated as a *bridge rule pattern*, as it stands for its versions obtained by substituting the designators with actual contexts' names. Bridge-rule instantiation may be performed by an agent also by means of bridge-update rules, that are in charge of replacing designators with actual suitable knowledge sources. We assume that bridge-update rules' conclusion $\beta(\hat{x})$ is in general a conjunction, possibly including actions of the following distinguished forms:

(i) $record(Item)$, which simply adds $Item$ to A's knowledge base; $Item$ can be either the "plain" bridge-rule result, or it can be obtained by processing such result via the evaluation of other atoms in $\beta(\hat{x})$);

(ii) $incorporate(Item)$, which performs some more involved elaboration for incorporating $Item$ into A's knowledge base. Notice that $incorporate$ is meant as a distinguished predicate, to be defined according to the specific application domain; in particular, it is intended to implement some proper form of belief revision.

(iii) $instantiate(S, m_i(k_i), L)$ which, for every bridge rule ρ with head matching with S, considers the context designator $m_i(k_i)$ and a list L of constants, and generates as many instances of ρ as obtained by substituting $m_i(k_i)$ (wherever it occurs) by elements of L. A bridge rules will be potentially applicable whenever all contexts in its body are constants, i.e., whenever all context designators, if present, have been replaced by actual contexts' names.

(iv) $enable(S, Q)$, which enables the application of a potentially applicable bridge rule ρ whose head matches with S and with associated trigger rule of the form Q **enables** S. It does so by generating its trigger, i.e., by adding Q as a new fact.

The combination of the introduction of both context designators and the $instantiate$ actions extends the expressiveness of the bridge-rule approach: even allowing variables

in place of contexts' names would not allow for the specific customization performed here. The purpose of defining context designators as terms is that of avoiding the requirement of the involved domains to be finite. In fact, context designators can denote values in an infinite domain, where, however, a finite number of *instantiate* actions generates a finite number of customized bridge rules. Notice that the computational complexity of the overall framework depends upon the computational complexity of the involved modules. In [8, 9] significant sample cases are reported.

5 Case Study: Bridge Rules Customization and Application

In order to explain the features that we have introduced so far we apply them to the case study. The agent acting on behalf of a prospective college student would for instance include the following trigger rule:

$$wish\_to\_enroll(Universities, Budget) \textbf{ enables}$$
$$chooseU(Universities, Budget, Selected\_UniversitiesL)$$

The meaning is that the agent is supposed to be able to conclude at some stage of its operation $wish\_to\_enroll(Universities, Budget)$, where $Universities$ is the list of universities which are of interest for the student, and $Budget$ is the budget which is available for completing the application procedure. Whenever this conclusion is reached, the trigger rule is proactively activated, thus enabling a suitable bridge rule. This bridge rule exploits a quantitative reasoning module and might correspond to this simple bridge rule pattern, where however there is the relative context designator $qr\_mod(mymod)$ to be instantiated.

$$chooseU(Universities, Budget, Selected\_UniversitiesL) \leftarrow$$
$$qr\_mod(mymod) : chooseU(Universities, Budget, Selected\_UniversitiesL)$$

Let us assume that the agent somehow (dynamically) instantiates this designator, e.g., to the name of a RASP module $rasp\_mod$, thus obtaining:

$$chooseU(Universities, Budget, Selected\_UniversitiesL) \leftarrow$$
$$rasp\_mod : chooseU(Universities, Budget, Selected\_UniversitiesL)$$

The RASP module, invoked via a suitable plugin, will return its results in $Selected\_UniversitiesL$, that will be a list representing the potential options for sending applications while staying within the given budget. A relevant role is performed by the corresponding bridge-update rule, which may have the form:

upon $chooseU(Universities, Budget, Selected\_UniversitiesL)$ **then**
$preferred\_subject(Subject)$,
$instantiate(apply(Univ, ResponseUniv), myuniv(u), Selected\_UniversitiesL)$,
$nearest\_sat\_center(Sc)$, $nearest\_toefl\_center(Tc)$,
$instantiate(general\_tests(Subject, R1, R2, R3), sat\_center(sc), [Sc])$,
$instantiate(general\_tests(Subject, R1, R2, R3), language\_center(lc), [Tc])$,
$enable(general\_tests(Subject, R1, R2, R3), enabledgentest)$

By evaluating the sub-queries from left to right, as it is usual in Prolog, this rule will determine the preferred subject $Subject$, and via an $instantiate$ action it will create

several copies of a bridge rule which finalizes the application (see below), namely one copy for each university included in $Selected\_UniversitiesL$. Notice that such bridge rules are not enabled yet. Then, the bridge-update rule finds the contexts' names Sc and Lc of nearest SAT and language-test centers respectively, where the student may perform the tests. The subsequent two $instantiate$ actions, together with the $enable$ action, will instantiate and trigger a suitable bridge rule pattern (shown below). The trigger part is in particular:

$$enabledgentest.$$
$$enabledgentest \textbf{ enables } general\_tests(Subject, R1, R2, R3)$$

which, as said, enables a bridge rule obtained by the following bridge rule pattern via its specialization to contexts' names Sc and Lc. This bridge rule will take care of performing the general tests, (among which the language certification) and filling the general part of the application.

$$\begin{aligned} general\_tests(Subject, R1, R2, R3) \leftarrow\ & sat\_center(sc) : general\_SAT\_test(R1), \\ & sat\_center(sc) : specific\_SAT\_test(R2), \\ & language\_center(lc) : language\_certification(R3), \\ & collegeorg : fill\_application \end{aligned}$$

Each test will return its results, which are then dynamically recorded, whenever available, by the bridge-update rule:

$$\textbf{upon } general\_tests(Subject, R1, R2, R3) \textbf{ then } record(test\_res(R1, R2, R3))$$

The recording of test results enables, via the following trigger rule, the application of the bridge rules, one for every selected university $Univ$, each of which will: send test the test results to that university; finalize the university-specific part of the application; wait for the response, returned in $ResponseUniv$.

$$\textbf{upon } test\_res(R1, R2, R3) \textbf{ then } apply(Univ, ResponseUniv)$$

The bridge rule pattern from which such bridge rules are obtained is:

$$\begin{aligned} apply(Univ, ResponseUniv) \leftarrow\ & test\_res(R1, R2, R3), \\ & myuniv(u) : send\_test\_results(R1, R2, R3), \\ & myuniv(u) : complete\_application(ResponseUniv) \end{aligned}$$

The corresponding bridge-update rules, of the form

$$\textbf{upon } apply(Univ, ResponseUniv) \textbf{ then } record(response(Univ, Response))$$

will record the responses, to allow a choice to be made among the universities that have returned a positive answer. Finally, enrollment must be finalized (code not shown here). Notice that, in the above bridge rules, some elements in the body implicitly involve the execution of specific actions (such as the payment of fees) that may take time to be executed, and may also involve user intervention (e.g., the student must personally and practically go to perform the SAT and TOEFL tests). Such actions have to be specified in the internal definition of the involved module(s), while user interventions emerge from the interaction between the agent and the user. For lack of space we do not discuss plan revision strategies (that might be needed in case of failure of some of the above steps), to be implemented via the agent's reactive and proactive features.

6 Semantics

In order to account for heterogeneity of composing modules, in MCSs and then in DACMACSs and in ACEs each module is supposed to be based upon a specific logic. Reporting from [8], a logic L is a triple $(KB_L; Cn_L; ACC_L)$, where KB_L is the set of admissible knowledge bases of L. A knowledge base is a set of KB-elements, or "formulas". Cn_L is the set of acceptable sets of consequences, whose elements are data items or "facts". Such sets can be called "belief sets" or simply "data sets". $ACC_L : KB_L \rightarrow 2^{Cn_L}$ is a function which defines the semantics of L by assigning to each knowledge-base a set of acceptable sets of consequences.

For any of the aforementioned frameworks, consider an instance $\mathcal{A} = \langle A_1, \ldots, A_h \rangle$ composed of h distinct modules, each of which can be either the basic agents, or an event-action module, or a reasoning module, or an external context. Each module is seen as $A_i = (L_i; kb_i; br_i)$ where L_i is a logic, $kb_i \in KB_{L_i}$ is the module's knowledge base and br_i is a set of bridge rules. A data state of $\mathcal{A}$ is a tuple $S = (S_1, \ldots, S_h)$ such that each of the S_is is an element of Cn_i, i.e. a set of consequences derived from A_i's knowledge base according to the logic in which module A_i is defined.

When modules are not considered separately, but rather they are connected via bridge rules, desirable data states, called *equilibria*, are those where bridge-rule application is considered. In MCSs, equilibria are those data states S where each S_i is acceptable according to function ACC_i associated to L_i, taking however bridge rules application into account. Technically, a data state S is an equilibrium iff, for $1 \leq i \leq n$, it holds that $S_i \in ACC_i(mng_i(app(S), kb_i))$. This means that if one takes the knowledge base kb_i associated to module A_i, considers all bridge rules which are applicable in data state S (i.e., S entails their body), applies the rules, applies the management function, it obtains exactly S_i (or at least S_i is one of the possible sets of consequences). Namely, an equilibrium is a data state that encompasses the application of bridge rules. In dynamic environments however, this does not in general imply that a bridge rule is applied only once, and that an equilibrium, once reached, lasts forever (conditions for reachability of equilibria are discussed in literature, see [23] and the references therein). In fact, contexts are in general able to incorporate new data items, e.g, as discussed in [10], the input provided by sensors. Therefore, a bridge rule is in principle re-evaluated whenever a new result can be obtained, thus leading to evolving equilibria.

As DACMACS and ACEs are frameworks for defining agents and multi-agent systems, the interaction with the external environment and with other agents goes beyond simple sensor input and must be explicitly considered. This is done by assuming, similarly to what is done in Linear Temporal Logic, a discrete, linear model of time where each state/time instant can be represented by an integer number. States $t_0, t_1, \ldots$ can be seen as time instants in abstract terms, though in practice we have $t_{i+1} - t_i = \delta$, where δ is the actual interval of time after which we assume a given system to have evolved.

Consider then a notion of *updates*: for $i > 0$, let $\Pi_i = \langle \Pi_{iA_1}, \ldots, \Pi_{iA_h} \rangle$ be a tuple composed of finite updates performed to each module and let $\Pi = \Pi_1, \Pi_2, \ldots$ be a sequence of such updates performed at time instants $t_1, t_2, \ldots$. Let $\mathcal{U}_E$, for $E \in \{A_1, \ldots, A_h\}$, be the *update operator* that each module employs for incorporating the new information, and let $\mathcal{U}$ be the tuple composed of all these operators. Notice that

each $\mathcal{U}_E$, i.e., each module-specific operator, encompasses the treatment of both self-generated updated and updated coming from interaction with an external environment.

In this more general setting data states evolve in time, where a *timed* data state at time T is a tuple $S^T = (S_1^T, \ldots, S_h^T)$ such that each S_i^T is an element of Cn_i at time T. The timed data state S^0 is an equilibrium according the MCSs definition. Later on however, transition from a timed data state to the next one, and consequently the definition of an equilibrium, is determined both by the update operators and by the application of bridge rules. A bridge rule ρ occurring in each composing module is now *potentially applicable* in S^T iff S^T entails its body. However, in the basic agent a potentially applicable bridge rule is applied only when it has been triggered by a trigger rule of the form seen above, i.e., if for some $T' \leq T$ we have that $S^{T'} \models Q$. In any event-action module M instead, a potentially applicable bridge rule is applied only if the module is *active*, i.e., if $S^{T'} \models tr_M$, where tr_M is an *event expression* which triggers the module evaluation (cf. [13]). Therefore, a timed data state of M at time $T+1$ is an equilibrium iff, for $1 \leq i \leq n$, it holds that $S_i^{T+1} \in ACC_i(mng_i(App(S^T), kb_i^{T+1}))$, where $kb_i^{T+1} = \mathcal{U}_i(kb_i^T, \Pi_T^i)$ and App is the extended bridge-rule applicability evaluation function. The meaning is that an equilibrium is now a data state which encompasses bridge rules applicability (with the new criteria) on the updated knowledge base. So, contexts now evolve in time, where we may say that $A_i^0 = (L_i; kb_i; br_i)$ as before, while $A_i^T = (L_i; kb_i^T; br_i)$. As discussed in [14], if both the update operators and the management functions preserve modules' consistency, then conditions for existence of an equilibrium (at some time T) are unchanged w.r.t. MCSs and DACMACS.

Notice that, for each bridge rule which is triggered (and so is applicable) at time T' the state when it is actually applied is not necessarily T', nor $T'+1$. In fact, a bridge rule becomes potentially applicable whenever a data state entail its body. So, the actual procedural sequence is the following:

- $S^{T'} \models Q$ for some trigger rule concerning bridge rule with conclusion $A(\hat{x})$, and then such a rule is executed at some time $T'' \geq T'$.
- At time $T \geq T''$ the results will be returned by the modules which are queried in the rule body; the case where $T' = T$, i.e., the bridge-rule body succeeds instantaneously, is an ideal extreme which is hardly the case in practice. In fact, internal and external modules may take some (a priori unpredictable) amount of time for returning their results.
- At time T, bridge-rule results will be elaborated by the management function, in our case implemented by the bridge-update rule.

The important aspect that allows us to smoothly incorporate enhanced ACE features in this semantics is that knowledge base updates in an agent are not necessarily determined from the outside. Rather, (part of) an update can also be the result of proactive self-modification. So, the generality and flexibility of ACE's semantics allows us to introduce advanced features without needing substantial modifications.

In particular, we consider bridge rule patterns as elements of the agent's knowledge base. A bridge rule pattern will produce new bridge rules only when its context designators will be instantiated. Such instantiation can be seen as a part of a self-modification, i.e, it can be seen as an update. Therefore, for the main agent we now have $A_i^0 = (L_i; kb_i; br_i)$ and $A_i^T = (L_i; kb_i^T; br_i^T)$, where at each subsequent time

the set of bridge rule associated to the module can be augmented by newly generated instances. The other definitions remain unchanged. This limited though effective semantic modifications constitute, in our opinion, a successful result of the research work that we present here. In fact, we obtain more general and flexible systems without significantly departing from the original MCSs' semantics, and this grants our approach a fairly general applicability.

7 Case Study: RASP Implementation

Below we discuss how to represent in RASP the case study discussed in Section 2. We do not report the full code, that the reader can find on the web site `http://www.dmi.unipg.it/formis/raspberry/` (section "Enrollment") where the solver Raspberry can also be obtained.[3] Our aim is to have a glance at how RASP works, and to demonstrate that the proposed approach is not only a more general architecture than basic ACE, but it has indeed a practical counterpart.

RASP code clearly must include a list of facts defining the universities to which the students is potentially interested, the SAT subjects (in general), and the SAT subjects corresponding to Courses (or Schools) available at each university.

```
% Universities
university(theBigUni).    university(theSmallUni).
university(thePinkUni).   university(theBlueUni).
university(theGreenUni).
% SAT subjects
sat_subject(literature).  sat_subject(mathematics).
sat_subject(chemistry).
% SAT subjects in each University
availableSubject(theBigUni, S) :- sat_subject(S).
availableSubject(theGreenUni, S) :- sat_subject(S).
availableSubject(theSmallUni, mathematics).
availableSubject(thePinkUni,  mathematics).
availableSubject(thePinkUni,  literature).
availableSubject(theBlueUni,  mathematics).
availableSubject(theBlueUni,  chemistry).
```

Below we then list: the tuition fees and the maximum fee allowed; the university rankings and the minimum required; the basketball team ranking, as it constitutes an additional evaluation factor.

```
% Tuition fees
tuitionFee(theBigUni,   21000).  tuitionFee(theSmallUni, 16000).
tuitionFee(thePinkUni,  15000).  tuitionFee(theBlueUni,  25000).
tuitionFee(theGreenUni, 15000).
% Constraint C1:  Tuition fee cannot exceed this threshold
maxTuition(22000).
```

[3] Raspberry, the grounder gringo (v.3.0.5), and the solver clasp (v.3.1.3) are used as follows:

```
raspberry_2.6.5 -pp -l3 -n 15000 -i enrollment_pref.rasp > enrollment_pref.asp
gringo-3.0.5 enrollment_pref.asp | clasp-3.1.3 0
```

```
% University reputation ranking R
reputation(theBigUni,   100).  reputation(theSmallUni,  90).
reputation(thePinkUni,   80).  reputation(theBlueUni,   75).
reputation(theGreenUni,  60).
% Constraint C2:  R must be higher than this threshold
reputationThrs(70).
% BasketballTeam Ranking
extraRank(theSmallUni, 10).  extraRank(theBigUni,   10).
extraRank(thePinkUni,   8).  extraRank(theBlueUni,   8).
extraRank(theGreenUni,  6).
```

The RASP fact below states that we have 1500 dollars, sum intended here as the budget available for completing applications. In general, symbol '#' indicates that an atom represents a resource. The constant before '#', here 'dollar', indicates the (arbitrary) name of the resource. The number after the '#' indicates an amount. In case of a fact, this amount is available initially, and can be then (in general) either consumed or vice versa incremented, as in RASP resource production can also be modeled.

```
% Budget for the application procedure
dollar#1500.
```

Now, the subject of interest and (if applicable) the status as foreign prospective students are indicated. Concerning the English language, nothing needs to be done if the student is not foreign, otherwise the TOEFL fee must be payed for performing the required test (we remind the reader that this RASP program evaluates the necessary expenses, so it is concerned with fees).

```
% My_subject
my_subject(mathematics).
% Omit the following fact if not foreign:
foreign.
% Language prerequisite
languageReqOK :- not foreign.
languageReqOK :- testTOEFLfee, foreign.
```

The universities where to potentially apply are derived according to the preferred subject, and the constraints concerning the university ranking and tuition fee. The student can apply if some university meeting the required requisites is actually found.

```
% Filtering of Universities
canApply(U,S) :- university(U), my_subject(S), reputation(U, R),
   availableSubject(U, S), reputationThrs(Th), R > Th,
   maxTuition(M), tuitionFee(U, Tu), Tu < M.
canApplyForSubject(Subj) :- canApply(Univ,Subj).
canApply :- canApply(Univ,Subject).
```

We now introduce proper RASP rules that perform quantitative reasoning, specifically by considering the fees for the different kinds of tests. The reader can ignore the prefix `[1-1]` which means that whenever the rule is applied, or "fired", this is done only once. This specification is not significant here, whereas it is useful in the description of more complex resource production/consumption processes.

```
% 1) General SAT test, fee1 fixed
  [1-1]: testSATfeeGen :- dollar#300, canApply.
% 2) Disciplinary SAT test, fee2 fixed
  [1-1]: testSATfeeSbj(mathematics) :-
            dollar#170, canApplyForSubject(mathematics).
  [1-1]: testSATfeeSbj(literature) :-
             dollar#180, canApplyForSubject(literature).
  [1-1]: testSATfeeSbj(chemistry) :-
            dollar#150, canApplyForSubject(chemistry).
  [1-1]: testSATfeeSbj(physics) :-
            dollar#160, canApplyForSubject(physics).
% 3) For foreign student, TOEFL fee3 fixed
  [1-1]: testTOEFLfee :- dollar#200, foreign, canApply.
% 4) Collegeorg application, fee4 fixed
  [1-1]: testCollegeOrg :- dollar#130, canApply.
```

A general rule with head `testGeneralDone` then establishes whether all general tests have been considered. If the available budget is too low and so no applications can issued, then no money is actually spent. Otherwise, the costs related to potential applications and the remaining amount (if any) are computed. Clearly, this code (omitted here) performs a quantitative evaluation and does not execute actual actions, which are left to the agent.

At this point, the Raspberry RASP solver can compute all solutions which maximize the number of applications. Solutions can be further customized with respect to the constraints. For instance, the standard `#maximize` ASP statements allow one to prefer universities with the best ranking and, in case of equivalent solutions, the ones with the best basketball team ranking (see the full code in the web site mentioned earlier, for the details on how to optimize the solution and enforce student preferences).

With the given facts, the best preferred solution provided by Raspberry involves applying to thePinkUni and theBigUni, with a total rating (sum of the two rankings) of 180 for the universities and 18 for the basketball teams.

If omitting maximization, there is a second solution which involves applying to thePinkUni and theSmallUni, with a total rating (sum of the two rankings) of 170 for the universities and 18 for the basketball teams.

The RASP module always returns the remaining (not spent) amount which is 90 dollars in the former case and 120 dollars in the latter one. Then, the agent might in general choose the best solution. However, it might instead choose another one based upon other criteria not expressed in the RASP program, i.e., geographic location or acceptance rates or maybe lesser expense, in case there would be relevant differences.

8 Concluding Remarks

The contribution of this paper is twofold. First, we have demonstrated, also by means of a practical example, how quantitative reasoning can be performed in agent-based frameworks. Second, we have enhanced modular approaches inspired to MCSs with partially specified bridge rules, that can be dynamically customized and activated according to the agent's reasoning results. The approach of this paper is fairly general,

and can be thus adapted to several application domains and to different agent architectures. Since no significant related work exists, our approach to coping with the cost of knowledge and the cost of action is relevant in a variety of domains, from logistics to configuration to planning, which are particularly well-suited for agents and MAS. An important application that we envisage is planning in robotic environments, where agents are embodied in robots that have limited resources available (first of all energy) and must complete their tasks within those limits, while possibly giving priority to the most important/urgent objectives.

References

[1] M. Alviano, C. Dodaro, W. Faber, N. Leone, and F. Ricca. WASP: A native ASP solver based on constraint learning. In P. Cabalar and T. C. Son, editors, *Proc. of LPNMR 2013*, volume 8148 of *LNCS*, pages 54–66. Springer, 2013.

[2] M. Bienvenu, J. Lang, and N. Wilson. From preference logics to preference languages, and back. In *Proc. of KR 2010*, pages 414–424, 2010.

[3] R. H. Bordini, L. Braubach, M. Dastani, A. E. Fallah-Seghrouchni, J. J. Gómez-Sanz, J. Leite, G. M. P. O'Hare, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica (Slovenia)*, 30(1):33–44, 2006.

[4] R. H. Bordini and J. F. Hübner. BDI agent programming in AgentSpeak using *Jason*. In F. Toni and P. Torroni, editors, *CLIMA VI, selected papers*, volume 3900 of *LNCS*, pages 143–164. Springer, 2006.

[5] A. Bracciali, N. Demetriou, U. Endriss, A. Kakas, W. Lu, P. Mancarella, F. Sadri, K. Stathis, G. Terreni, and F. Toni. The KGP model of agency: Computational model and prototype implementation. In *Global Computing: IST/FET Intl. Workshop*, LNAI 3267, pages 340–367. Springer, 2005.

[6] G. Brewka, J. P. Delgrande, J. Romero, and T. Schaub. asprin: Customizing answer set preferences without a headache. In B. Bonet and S. Koenig, editors, *Proc. of AAAI-15*, pages 1467–1474. AAAI Press, 2015.

[7] G. Brewka and T. Eiter. Equilibria in heterogeneous nonmonotonic multi-context systems. In *Proc. of AAAI-07*, pages 385–390. AAAI Press, 2007.

[8] G. Brewka, T. Eiter, and M. Fink. Nonmonotonic multi-context systems: A flexible approach for integrating heterogeneous knowledge sources. In M. Balduccini and T. C. Son, editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, volume 6565 of *LNCS*, pages 233–258. Springer, 2011.

[9] G. Brewka, T. Eiter, M. Fink, and A. Weinzierl. Managed multi-context systems. In T. Walsh, editor, *Proc. of IJCAI 2011*, pages 786–791. IJCAI/AAAI, 2011.

[10] G. Brewka, S. Ellmauthaler, and J. Pührer. Multi-context systems for reactive reasoning in dynamic environments. In T. Schaub, editor, *Proc. of ECAI-14*. IJCAI/AAAI, 2014.

[11] G. Brewka, I. Niemelä, and M. Truszczyński. Preferences and nonmonotonic reasoning. *AI Magazine*, 29(4), 2008.

[12] A. Brik. *Extensions of Answer Set Programming*. PhD thesis, University of California, San Diego, 2012.

[13] S. Costantini. ACE: a flexible environment for complex event processing in logical agents. In M. Baldoni, L. Baresi, and M. Dastani, editors, *EMAS-15, Revised Selected Papers*, volume 9318 of *LNCS*. Springer, 2015.

[14] S. Costantini. Knowledge acquisition via non-monotonic reasoning in distributed heterogeneous environments. In M. Truszczyński, G. Ianni, and F. Calimeri, editors, *Proc. of LPNMR-13*, volume 9345 of *LNCS*. Springer, 2015.

[15] S. Costantini and G. De Gasperis. Exchanging data and ontological definitions in multi-agent-contexts systems. In A. Paschke, P. Fodor, A. Giurca, and T. Kliegr, editors, *Proc. of RuleML 2015 Challenge*, CEUR Workshop Proceedings. CEUR-WS.org, 2015.
[16] S. Costantini and A. Formisano. Modeling preferences and conditional preferences on resource consumption and production in ASP. *Journal of of Algorithms in Cognition, Informatics and Logic*, 64(1), 2009.
[17] S. Costantini and A. Formisano. Answer set programming with resources. *Journal of Logic and Computation*, 20(2):533–571, 2010.
[18] S. Costantini and A. Formisano. RASP and ASP as a fragment of linear logic. *Journal of Applied Non-Classical Logics*, 23(1-2):49–74, 2013.
[19] S. Costantini, A. Formisano, and D. Petturiti. Extending and implementing RASP. *Fundam. Inform.*, 105(1-2):1–33, 2010.
[20] S. Costantini and A. Tocchio. A logic programming language for multi-agent systems. In *Proc. of JELIA-02*, volume 2424 of *LNAI*. Springer, 2002.
[21] S. Costantini and A. Tocchio. The DALI logic programming agent-oriented language. In *Proc. of JELIA-04*, volume 3229 of *LNAI*. Springer, 2004.
[22] A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. GASP: answer set programming with lazy grounding. *Fundam. Inform.*, 96(3):297–322, 2009.
[23] M. Dao-Tran, T. Eiter, M. Fink, and T. Krennwallner. Distributed evaluation of nonmonotonic multi-context systems. *JAIR*, 52:543–600, 2015.
[24] M. Dastani, M. B. van Riemsdijk, and J. C. Meyer. Programming multi-agent systems in 3APL. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 39–67. Springer, 2005.
[25] J. Delgrande, T. Schaub, H. Tompits, and K. Wang. A classification and survey of preference handling approaches in nonmonotonic reasoning. *Computational Intelligence*, 20(12):308–334, 2004.
[26] A. Dovier, A. Formisano, E. Pontelli, and F. Vella. A GPU implementation of the ASP computation. In *Proc. of PADL 2016*, volume 9585 of *LNCS*, pages 30–47. Springer, 2016.
[27] M. Fisher. MetateM: The story so far. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *PROMAS*, volume 3862 of *LNCS*, pages 3–22. Springer, 2005.
[28] M. Gebser, R. Kaminski, B. Kaufmann, J. Romero, and T. Schaub. Progress in clasp series 3. In M. Truszczyński, G. Ianni, and F. Calimeri, editors, *Proc. of LPNMR-15*, volume 9345 of *LNCS*, pages 368–383. Springer, 2015.
[29] M. Gelfond. Answer sets. In *Handbook of Knowledge Representation*. Elsevier, 2007.
[30] K. V. Hindriks, W. van der Hoek, and J. C. Meyer. GOAL agents instantiate intention logic. In A. Artikis, R. Craven, N. K. Cicekli, B. Sadighi, and K. Stathis, editors, *Logic Programs, Norms and Action*, volume 7360 of *LNCS*, pages 196–219. Springer, 2012.
[31] G. Liu, T. Janhunen, and I. Niemelä. Answer set programming via mixed integer programming. In *Proc. of KR 2012*, 2012.
[32] M. Maratea, L. Pulina, and F. Ricca. A multi-engine approach to answer-set programming. *TPLP*, 14(6):841–868, 2014.
[33] P. Naumov and J. Tao. Budget-constrained knowledge in multiagent systems. In G. Weiss, P. Yolum, R. H. Bordini, and E. Elkind, editors, *Proc. of AAMAS 2015*, pages 219–226. ACM, 2015.
[34] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002.
[35] Web-references. Some ASP solvers. Clasp: `potassco.sourceforge.net`; Cmodels: `www.cs.utexas.edu/users/tag/cmodels`; DLV: `www.dlvsystem.com`; Smodels: `www.tcs.hut.fi/Software/smodels`.

Monitoring Patients with Hypoglycemia using Self-Adaptive Protocol-Driven Agents: a Case Study

Angelo Ferrando, Davide Ancona and Viviana Mascardi

[1]DIBRIS, University of Genova, Italy
angelo.ferrando@dibris.unige.it, davide.ancona@unige.it,
viviana.mascardi@unige.it

Abstract. Trace expressions are a compact and expressive formalism, originating from our previous research on global types and constrained global types, for specifying complex patterns of actions. Global types were initially devised for runtime verification of agent interactions in multiagent systems and have been successfully employed to model real protocols and to generate monitors for the Jason and JADE platforms. Recently, their extension with constraints (constrained global types) has been exploited to model interaction protocols which can drive the agents' behavior, leading to "protocol-driven agents" whose procedural knowledge is given by the protocol, and whose behavior results from the protocol interpretation. In this paper we discuss how we can model medical protocols as trace expressions in an e-Health scenario. For this purpose, we have extended our previous work on protocol-driven agents by allowing agents to be guided by patterns of actions specified as trace expressions instead of constrained global types, and by allowing events in the trace expression to be of any kind instead of communicative ones only.

Key words: Trace expressions, Protocol-Driven Agents, e-Health protocol, Protocol consistency, Hypoglycemia in newborns, Patient monitoring

1 Introduction and Motivation

The demographic changes of our societies are causing an explosion of care requests and, as a consequence, of the healthcare expenses.

Care requests for minor problems that could be addressed without the direct intervention of the doctor divert the healthcare resources from more serious situations. One possible solution to this problem, addressed since the beginning of the millennium, is *Remote Patient Monitoring* (RPM, [7]) which consists in the remote and distributed monitoring of a specific category of patients in order to limit the number of visits to doctors and hospitals.

Health telematics can play a major role in improving the lives of patients [13], particularly in the weaker sections of the society including disabled, elderly and chronically ill patients [23]. Mobile health-monitoring devices offer great help for

such patients who may afford good healthcare without having to regularly visit their doctor. These technologies bring potential benefits to both the patient and the doctor; doctors can focus on priority tasks by saving time normally spent with consulting chronically ill patients, and patients can be properly looked after whilst remaining in their environment without having to make tiring and time-consuming visits.

From a technological point of view, RPM requires a low level physical infrastructure made up of sensors and a software middleware monitoring the sensors output and implementing rules for warning either the patient or the doctor, or both, if the pattern of perceived data diverges from the "normal" pattern for that specific patient. In order to achieve its goals, the middleware should:

1. manage data coming from decentralized and heterogeneous sensors;
2. dynamically plug and remove sensors and other components;
3. be adaptive to changes taking place in the environment and in the care protocols that must be adopted;
4. interact with the human beings involved in the RPM process;
5. be fault tolerant.

A multiagent system with one agent in charge for each patient and one for each doctor involved in the RPM process seems a very natural choice to satisfy all these requirements, due to the MAS intrinsic structure where each agent has incomplete information or capabilities for solving the problem, there is no global system control, data is decentralized, computation is asynchronous, the system is open and highly dynamic. Consistently with a holonic approach to MAS engineering and development [18], the agent in charge of the patient (Patient's Agent, PA in the sequel) might be a MAS as well, with one agent in charge for each sensor, one agent in charge for the user interface, one agent in charge for the identification of threats based on sensory input, and so on.

The PA should monitor the patient's health dynamics by verifying that it follows a known "protocol", should implement those actions foreseen by the protocol (for example, "if the blood pressure is below a given threshold, tell the patient to sit down for 10 minutes, switch the saturation sensor on, and increase the heartbeat monitoring frequency"), and should quickly identify deviations from the followed protocol. Since the protocol guiding the patient's treatment can change over time, the PA should be able to dynamically move from one protocol to another, upon request of some trusted entity like the doctor. This protocol switch might take place also when sensory input shows a critical situation and some "higher severity" protocol must be immediately adopted. To summarize, the PA should be driven by the protocol and should be able to adapt to new situations by changing the followed protocol when needed.

In our recent research [2] we designed and implemented a framework for protocol-driven self-adaptive agents, where agents are characterized by one interaction protocol specified using constrained global types [1, 4] and by three mandatory components, the *knowledge base*, the *message queue* and the *environment's representation*, that should be directly implemented in the underlying agent framework. Being protocol-driven means that the agent behaves according

to a given protocol. In each time instant, the protocol-driven agent can make only those internal choices which are allowed by the protocol in the current state. In case of events which depend on external choices, the agent can only verify if the event that took place is compliant with the protocol and act consequently. We do not generate any agent code into any agent-oriented programming language. The protocol specification is interpreted and this gives the flexibility that meta-programming ensures, demonstrated for example by the easiness in implementing protocol switch: protocols can be exchanged and modified at run-time, being first class entities.

In this paper we extend that framework by allowing protocols to involve events of any kind and not just communicative ones. The extended framework is made more usable by allowing protocols to be expressed using "trace expressions" [5]. We show how the extended framework can be profitably adopted to specify medical protocols and to monitor them. We believe that our framework, running on top of JADE and Jason, can serve as the basis for implementing the RPM software middleware. The motivating scenario that we consider is that of newborns who may suffer from hypoglycemia and the protocols we have implemented are based on medical literature [21].

The paper is organized in the following way: Section 2 discusses the related work, Section 3 introduces the background knowledge for understanding the protocol modeling and the experiments presented in Section 4, and Section 5 concludes.

2 Related work

In the last years, the exploitation of multiagent systems in the e-Health scenario has become more and more widespread, as discussed for example in [9]. E-Health systems pose many challenges and requirements, such as:

- context and location awareness are to be smoothly integrated, i.e., the access and the visualization of health-related information always depends on the overall contexts of the patient and of the user [12],
- fault-tolerance, reliability, security and privacy-awareness are a must in order to accommodate the strict requirements of all healthcare applications,
- effective mobile devices are to be used to provide access to relevant health-related information independently of the current physical location and physical condition of the user, and
- unobtrusive sensor technology is needed to gather the physiological information from the patient without hampering her daily life.

All mentioned requirements immediately recall the characterizing features of multiagent systems and it comes with no surprise that many ubiquitous and pervasive e-Health systems are designed and realized using multiagent abstractions and technologies [6, 19, 24].

According to [16], multiagent systems in the e-Health context are used in the following categories of applications: *assistive living*; *diagnosis*; *physical monitoring*; and *smart-emergency*. The scenario we take under consideration in this

paper falls in the third category, where the aim is the continuous monitoring of patients at home [20].

In [13], a system architecture consisting in a Java-based agent for each human role (e.g. doctors, patients) is presented. Within the framework, agents reside in three areas: the patient's mobile device (e.g. smart phone or PDA with Internet connectivity); the healthcare personnel's mobile device (e.g. for nurses or paramedics); and the mobile and static servers (which may be a wireless connected notebook or an enterprise server computer). Our work is close to that approach since we have one protocol-driven agent for each human role as well.

A framework for representing in formal terms how clinical guidelines[1] are realized through the actions of individuals organized into teams is presented in [25]. The authors emphasize that the flexibility to deviate from the guideline recommendations is indispensable if we are to build workflow systems that will be accepted by the medical community. Even if the aim of our work is different from that discussed in [25], they share some ideas, in particular regarding the reuse of plans (which, in our approach, are protocols), the team based definition ("global protocol definition" in our approach), and the flexibility of changing protocol at runtime.

In [11, 14] the authors describe the GPROVE framework for specifying medical guidelines[2] in a visual way and for verifying the conformance of the guideline execution w.r.t. the specification. Except for the part regarding the visual representation of guidelines, that work is very close to ours, in particular as far as the verification that generated events do not lead to discrepancies with the models is concerned. The most relevant difference is when verification is performed: in [11, 14] the verification is performed on a log file generated during the execution (*a posteriori*). In our work it is done at runtime: as soon as an event relevant for the application is intercepted, the agent interpreter verifies if it is compliant with the protocol and selects the most suitable action to take, according to the verification outcome and to the selection and reaction strategies that the agent implements.

Another work similar to ours is [22] where the SUAP project, a MAS to support and monitor prenatal care, is presented. SUAP manages electronic healthcare records of pregnant women; models, monitors and provides advice on prenatal protocols; and models a simple referencing protocol based on pregnancy risks. In that work there are *protocol agents* which monitor data related to appointments and exam results to identify situations in which protocols must be applied (protocols are defined by a set of rules). In [22] protocols can not change during the execution: this represents the main different of that work w.r.t. ours.

As discussed in [17], multimedia delivery in e-Health systems is characterized by a wide spectrum of dynamically varying Quality of Service (QoS) requirements which must be negotiated, re-negotiated and managed in response

[1] Clinical guidelines are special types of plans realized by collective agents.

[2] Medical guidelines are clinical behavior recommendations used to help and support physicians in the definition of the most appropriate diagnosis and/or therapy within determinate clinical circumstances.

to changing network and end-system conditions, or to new expectations from the human user. Thus in an e-Health context, it is precisely this (re)negotiation and dynamic management of applications QoS that emphasizes the need for adaptable protocols. It is therefore clear that any new solution, which attempts to efficiently deal with the problem of e-Health QoS provisioning, must be adaptive. Even if the work discussed in [17] is about the QoS context, it allows focusing on a real problem that all e-Health systems have to contend, which is the versatility and the ability to change quickly. Our framework can tackle the QoS problem, as each agent inside the system is able to change its protocol as a consequence of a protocol switch request sent by a privileged agent; in this way, we can provide the correct protocol to follow in each situation: the protocol switch derives from the observation of the current state of the system.

3 Self-Adaptive Protocol-Driven Agents

In [2] we presented a framework for implementing protocol-driven agents whose internal architecture is depicted in Figure 1.

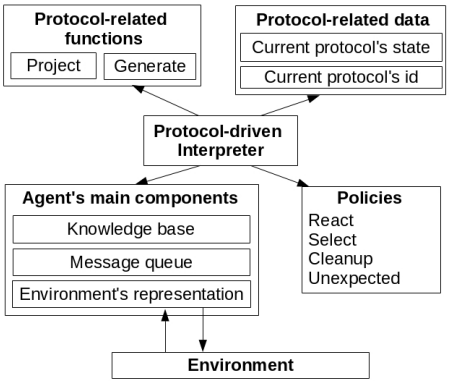


Fig. 1. Architecture of a protocol-driven agent.

In this section we summarize that work and we present some extensions which make the framework more general and flexible, consistently with the extensions to the protocol formalism described in [5]. In [2], in fact, the protocol could involve communicative actions only and hence could be used to drive (and, at the same time, to monitor) the communicative behavior of the agents. In this paper we move a step forward, changing the framework implementation to cope with protocols where any kind of perceived events can be modeled.

Being protocol-driven means that the agent behaves according to a given protocol, or, more in general, a given "pattern of events and actions". As already introduced in Section 1, the protocol-driven agent can only make those internal

choices which are allowed by the protocol in the current state, and – in case of perceived events – it can only verify if the event is compliant with the protocol. This architecture supports a combined approach to correct behavior generation (when the choice of what to do is up to the agent) and runtime monitoring (when the choice of what to do is made by the agent environment, and the agent must verify that it is protocol-compliant).

For supporting a protocol-driven approach to agent programming, a formalism for expressing protocols must exist together with a *generate* function for identifying the allowed actions for moving from the current state of the protocol to the next one. What differentiates the behavior of each agent are the *select* policy to select the action to perform among the allowed ones, and the *react* policy to react to perceived events. Two more policies must be defined to state how to manage *unexpected* events and which *cleanup* actions to perform before switching from the currently executing protocol to the new one. Protocol switch is one of the major features of our approach, allowing agents to self-adapt to new situations by changing the current protocol upon reception of "switch requests". Those agents that have the power to cause a protocol switch must be explicitly stated by the agent and may change over time. Each agent can change the currently executing protocol by requesting a protocol switch to itself. Each agent Ag is also able to project a global description of a protocol involving many agents onto a local version by keeping only events that involve Ag itself. If a description of the global protocol that all the agents in the MAS must respect exists, the local protocol for each agent can be automatically obtained from the global one. This allows *the whole MAS to respect the global protocol by construction*, as the local versions are obtained from the global one by projection, and are consistent with it.

Trace expressions. Trace expressions are a specification formalism expressly designed for runtime verification; they are an evolution of global types [4] and have been continuously refined and consolidated during the last 4 years [1, 3, 5, 15]. Trace expressions build on top of the "event" notion. An event may either be a communicative event, like in [2], or any other event which may take place in the environment and that the agent can perceive.

Events. In the following we denote by $\mathcal{E}$ a fixed universe of events. An event trace over $\mathcal{E}$ is a possibly infinite sequence of events in $\mathcal{E}$. A trace expression over $\mathcal{E}$ denotes a set of event traces over $\mathcal{E}$.

Event types. To be more general, trace expressions are built on top of event types (chosen from a set $\mathcal{ET}$), rather than of single events; an event type denotes a subset of $\mathcal{E}$ and can be expressed by means of the `has_type` predicate. An example is provided in Section 4.

Trace expressions. A trace expression τ represents a set of possibly infinite event traces and is defined on top of the following operators:

- ϵ (empty trace), denoting the singleton set $\{\epsilon\}$ containing the empty event trace ϵ (the empty trace is represented by the `lambda` symbol in the actual Jason code).

- $\vartheta{:}\tau$ (*prefix*), denoting the set of all traces whose first event e matches the event type ϑ ($e \in \vartheta$), and the remaining part is a trace of τ (: symbol in the code).
- $\tau_1 \cdot \tau_2$ (*concatenation*), denoting the set of all traces obtained by concatenating the traces of τ_1 with those of τ_2 (* symbol).
- $\tau_1 \wedge \tau_2$ (*intersection*), denoting the intersection of the traces of τ_1 and τ_2 (/\ symbol).
- $\tau_1 \vee \tau_2$ (*union*), denoting the union of the traces of τ_1 and τ_2 (\/ symbol).
- $\tau_1 | \tau_2$ (*shuffle*), denoting the set obtained by shuffling the traces in τ_1 with the traces in τ_2 (| symbol).

To support recursion without introducing an explicit construct, trace expressions are regular (a.k.a. rational or cyclic) terms and can be represented by a finite set of syntactic equations, as happens, for instance, in most modern Prolog implementations where unification supports cyclic terms.

As an example, $T = \vartheta{:}T$ represents the infinite but regular term $\vartheta{:}\vartheta{:}\vartheta{:}\vartheta{:}\ldots$. The lack of a base case for this recursive definition is not a problem, as trace expressions are interpreted in a coinductive way in order to represent infinite traces of events like this one, besides finite ones.

The semantics of trace expressions is specified by the transition relation $\delta \subseteq \mathcal{T} \times \mathcal{E} \times \mathcal{T}$, where $\mathcal{T}$ denotes the set of trace expressions. As it is customary, we write $\tau_1 \xrightarrow{e} \tau_2$ to mean $(\tau_1, e, \tau_2) \in \delta$. If the trace expression τ_1 specifies the current valid state of the system, then an event e is considered valid iff there exists a transition $\tau_1 \xrightarrow{e} \tau_2$; in such a case, τ_2 will specify the next valid state of the system after event e. Figure 2 defines the rules for the transition function (not all the rules are shown for space constraints; in particular, or and shuffle rules have a symmetric or-r and shuffle-r version where τ_2 moves, instead of τ_1).

$$\text{(prefix)}\frac{}{\vartheta{:}\tau \xrightarrow{e} \tau}\ {\scriptstyle e \in \vartheta} \qquad \text{(or-l)}\frac{\tau_1 \xrightarrow{e} \tau_1'}{\tau_1 \vee \tau_2 \xrightarrow{e} \tau_1'} \qquad \text{(and)}\frac{\tau_1 \xrightarrow{e} \tau_1' \quad \tau_2 \xrightarrow{e} \tau_2'}{\tau_1 \wedge \tau_2 \xrightarrow{e} \tau_1' \wedge \tau_2'}$$

$$\text{(shuffle-l)}\frac{\tau_1 \xrightarrow{e} \tau_1'}{\tau_1 | \tau_2 \xrightarrow{e} \tau_1' | \tau_2} \qquad \text{(cat-l)}\frac{\tau_1 \xrightarrow{e} \tau_1'}{\tau_1 \cdot \tau_2 \xrightarrow{e} \tau_1' \cdot \tau_2} \qquad \text{(cat-r)}\frac{\tau_2 \xrightarrow{e} \tau_2'}{\tau_1 \cdot \tau_2 \xrightarrow{e} \tau_2'}\ {\scriptstyle \epsilon(\tau_1)}$$

Fig. 2. Operational semantics of trace expressions. The $\epsilon(\tau)$ side condition means that τ can move into the empty trace expression.

Template trace expressions. In order to write complex protocols in a compact and readable way, in [15] we extended our previous work by introducing templates which allow parameters inside the protocol definition. In this section we introduce template trace expressions which extend the notion of template global types.

Template trace expressions are a "meta-formalism": they must be applied to some arguments in order to obtain "normal" trace expressions. Parameters

are present only in the template trace expression definition: when template trace expression are used either for runtime verification or for protocol driven behavior generation, all terms must be ground.

Let us consider the following template trace expressions, modeling infinite loops of request reception and management, which must be composed using the *shuffle* (|) operator as many times as the number of values over which `var(1)` will vary:

```
SERVERT = receive_request(var(1)): (serve_request(var(1)): SERVERT),
SERVER = finite_composition(|, SERVERT, [var(1)])
```

A template trace expression must be "applied" in order to turn into a normal trace expression, as shown in the code fragment below:

```
..., SERVER = finite_composition(|, SERVERT, [var(1)]),
apply(SERVER,[t(var(1),[client1,client2,client3])],INSTANTIATEDSERVER), ...
```

The *apply* predicate instantiates the `SERVER` protocol unifying its instantiation with the `INSTANTIATEDSERVER` variable, whose actual value will become:

```
SERVER1 = receive_request(client1):(serve_request(client1):SERVER1),
SERVER2 = receive_request(client2):(serve_request(client2):SERVER2),
SERVER3 = receive_request(client3):(serve_request(client3):SERVER3),
INSTANTIATEDSERVER = SERVER1|(SERVER2|SERVER3)
```

The value associated with `INSTANTIATEDSERVER` is a ground trace expression which can be projected and used for runtime monitoring and protocol-driven behavior.

The great advantage of using template trace expressions is that they are more compact and easy to design and understand than their "unfolding", and that the set over which the variables range can be decided at runtime, hence allowing the agents to implement a limited form of *dynamic protocol generation*.

Implementation details. With respect to the implementation described in [2], we made minor changes to the protocol-driven agent interpreter and to the project function to cope with the presence of generic events in the protocol, instead of communicative events only. The new implementation of our protocol-driven agents runs on top of both Jason [10] and JADE [8]. The agent interpreter is driven by the δ transition function described before and is implemented in Prolog. Since Jason can directly integrate Prolog code, the interpreter has been easily embedded into Jason agents. As far as JADE is concerned, we used a bidirectional Java-Prolog interface to make JADE agents behave according to Prolog rules.

The transition function is implemented by Prolog clauses which are in a one-to-one correspondence with the δ transition rules, for a total of about 20 LOC (Lines Of Code). The projection algorithm amounts to 60 lines of Prolog code, and the management of templates and their application required less than 200 LOC. Besides these Prolog predicates which are independent of the underlying framework, we had to develop some ad-hoc code both for JADE and for Jason, for linking the interpreter code with the agents behavior. In both cases, the required LOCs were less than 150.

4 Modeling Medical Protocols for Hypoglycemia in the Newborns

In the scenario we address as a case study, we have many doctors and many patients (newborns suffering from hypoglycemia). Each human being involved in the scenario, namely the doctors and the newborns, is associated with a protocol-driven agent. We assume that each baby can be equipped with sensors able to change the protocol-driven agent's knowledge base after perception of sensory input such as *temperature*, *heartbeat*, *pressure*, O_2 *saturation*, *movement*, and so on. For example, if the patient has tremors, the movement sensor will update the knowledge base of the protocol-driven agent with information about the perceived tremors; this change in the agent knowledge base will "fire" a move into a new protocol state where the successive foreseen events are those which are normally expected when the newborn has tremors. If the perceived event is instead an exceptional one and raises an emergency, a protocol switch might take place for allowing the agent to abandon the normal protocol it is following, and adopt an exceptional one suitable for managing the exceptional event. Otherwise, if the event is unknown, the agent will start following its own *unexpected events* policy. Event perception may cause the agent to perform other actions, defined by its *react* policy. Besides defining which events are expected in a given state, the protocol also asserts which actions are allowed in a given state. If the allowed actions are more than one, the agent will select one among them by using its *select* policy[3].

The protocol-driven agent associated with newborns might need to communicate something to the patient or, to be more precise, with the parents of the patient. To achieve this goal it may print some message onto a screen positioned near to the newborn. The parents can follow the monitoring process and the intervention instructions like, for example, the request to inject a dose of glucose solution. Hypoglycemia management does not require a constant presence of a doctor but needs an ongoing monitoring of some vital parameters, falling in the "patient monitoring at home" scenario discussed in Section 1.

For sake of readability, in the sequel we will consider the situation where there are *1* doctor and N patients (Figure 3), even if the protocols can be easily generalized to M doctors.

The doctor's agent is driven by a single protocol while the patients' agents may be driven, in each time instant, by one of the three protocols below:

- *standard protocol*, which models the situation where the newborn has no symptoms of the disease;
- *severity 1 protocol*, associated with the lowest severity level of the disease;
- *severity 2 protocol*, associated with the highest severity level of the disease.

[3] In our prototypical implementation, these policies are the default ones: unexpected events are discarded, no reaction is associated with perceived events besides that hard-wired into the protocol, and selection selects the first action returned by the *generate* function.

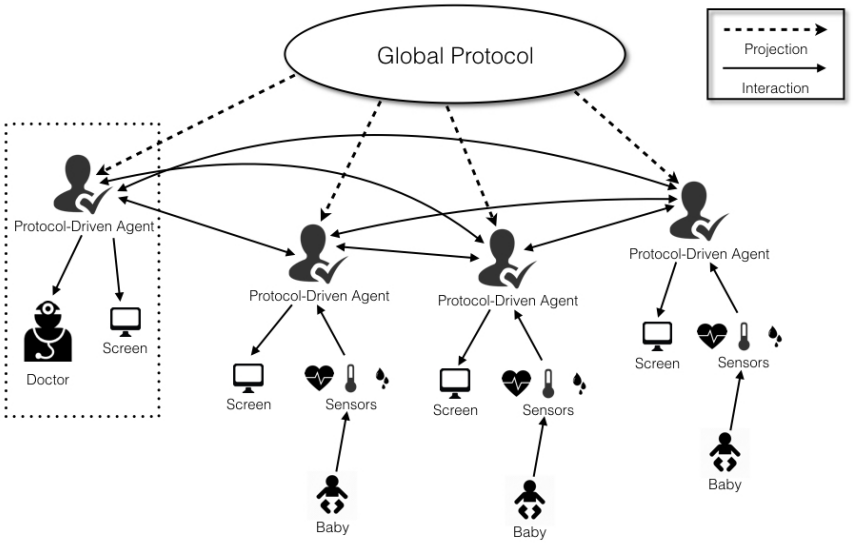


Fig. 3. Doctor-Patients architecture.

Figure 4 shows the cases when the agent associated with the newborn can continue its execution, and when it must perform a protocol switch. Protocol switches are fired by reception of a switch request from a trusted agent. Each agent may send a switch request to itself upon perception of an exceptional event, and this allows us to model in a neat and simple way event perception as a trigger of a protocol switch. This feature allows us to write protocols depending for example on

- a change in the agent's knowledge base (caused for instance by an event generated by a sensor), or
- a change in the environment, perceived by all the agents immersed in it.

Doctor protocol. The protocol `DP` of the agent associated with the doctor is conveniently represented by a template trace expression (the trace expression associated with the `DoctorPatientProtocol` logical variable, which will be replicated as many times as necessary during the application stage) consisting of the union (`\/` operator) of three sub-protocols describing three mutually exclusive situations that the agent can experience. The event types appearing in the protocol may be communicative ones. In this case, for sake of readability, we prefix their name by `msg`. The first argument of communicative event types will be instantiated with the sender of the message and the second with the receiver. The agent associated with the doctor has the power to request a protocol switch to the patient's agent. A protocol switch in the patient may take place also in

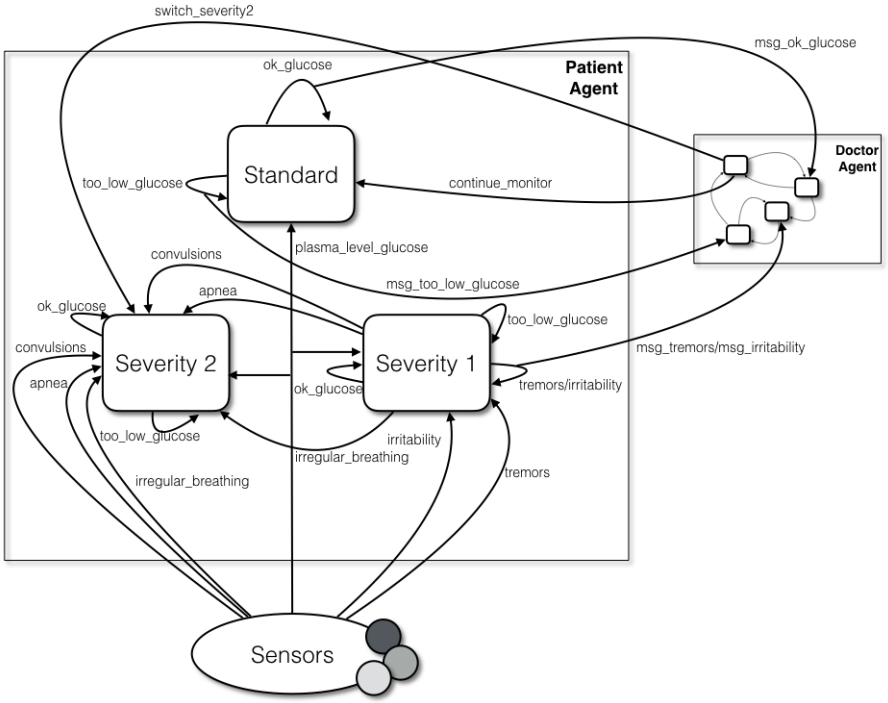


Fig. 4. Patient's protocols execution.

response to sensory input, as described later. A switch request is modeled by the `switch_request(Sender,Receiver,NewProtocolIdentifier)` event type.

- `Ok` sub-protocol: if the doctor agent receives a message reporting that the glucose level in the blood of its patient is ok, then things are going on in the right way: the agent moves to the `DoctorPatientProtocol` state again (`:` operator, modeling sequence) and continues to monitor;
- `SevereProblem` sub-protocol: if the doctor agent receives a message reporting that the glucose level in the blood of its patient is too low, then it sends a protocol switch request to the patient (`switch_request(var(doctor), var(patient), severity2)` bringing the patient's protocol to `severity2`) and continues to monitor (namely, it moves to the `DoctorPatientProtocol` again thanks to the concatenation operator, `*`);
- `SwitchedToSeverity1` sub-protocol: if the doctor agent receives a message reporting that the patient has switched its protocol to `severity1` (communicative event type `msg_switched_to_severity1(var(patient),var(doctor))`), then the situation requires a more careful monitoring as it might change into a severe health status:
  - if the agent either receives a message reporting that the patient has tremors or she is irritable (`TremorsOrIrritability` sub-protocol), then it

either sends a protocol switch request bringing the patient's protocol to `severity2`, or it asks the patient's agent to continue to monitor without switching to the `severity2` protocol;
- if, before or after the messages related to either tremors or irritability (`|` models the occurrence of events in any order), the doctor's agent receives a message from the patient (or from his parents) reporting an intervention request (`InterventionRequest` sub-protocol), then the agent communicates this request to the doctor using the screen.

In both cases, the agent can move to the "standard" monitoring state modeled by the trace expression associated with `DoctorPatientProtocol`.

```
trace_expr_template(doctor_protocol, DP) :-
DoctorPatientProtocol =  Ok \/ SevereProblem \/ SwitchedToSeverity1,
Ok = msg_ok_glucose(var(patient),var(doctor)):DoctorPatientProtocol,
SevereProblem =
     msg_too_low_glucose(var(patient),var(doctor)):SwitchToSeverity2,
SwitchedToSeverity1 = (msg_switched_to_severity1(var(patient),var(doctor))
    :(TremorsOrIrritability|InterventionRequest))*DoctorPatientProtocol,
SwitchToSeverity2 =
    (switch_request(var(doctor),var(patient),severity2) \/
     msg_continue_monitor(var(doctor),var(patient)))*DoctorPatientProtocol,
TremorsOrIrritability =
    (msg_tremors(var(patient),var(doctor)):lambda \/
     msg_irritability(var(patient),var(doctor)):lambda)*SwitchToSeverity2,
InterventionRequest =
     msg_intervention_request(var(patient),var(doctor)):
     print_intervention_request(var(patient),var(interface)):lambda,
DP = finite_composition(|, DoctorPatientProtocol,
     [var(patient),var(doctor),var(interface)]).
```

During the application stage, `var(patient)` will vary on the doctor's patients (for example `[patient1, patient2, patient3]`), `var(doctor)` varies on the doctors involved in the monitoring (for example, `doctor1` if we want to keep the scenario as simple as possible) and `var(interface)` varies on the artifacts in the MAS that can act as an interface between the humans involved in the loop and the system. For example, we might decide that messages that the patients must see are printed on a screen named `screen`. We could also send messages to more than one artifact, for example to `[local-screen, doctor-mobile-phone, father-mobile-phone, mother-email]`[4]. The code fragment for the instantiation of the template trace expression above is

```
apply(T, [t(var(patient), [patient1, patient2, patient3]),
          t(var(doctor), [doctor1]),
          t(var(interface), [screen2])],
      INSTANTIATEDPROTOCOL)
```

[4] In the Jason implementation discussed later on, we modeled these artifacts as "dumb" agents which can receive FIPA-ACL messages. This was an easy and quick way to build a working prototype including all the relevant MAS components, without needing to actually implement Java classes for the artifacts in the system.

Patient standard protocol. The `StandardProtocol` trace expression models three situations, corresponding to three sub-protocols:

- `Ok` sub-protocol, modeling the normal situation where the perceived glucose level is ok (event type `ok_glucose(var(patient))`, representing a perception event); the doctor is informed (`msg_ok_glucose(var(patient),var(doctor))`) and the protocol moves to the situation modeled by `StandardProtocol`.
- `TooLowGlucose` sub-protocol: the perceived event has type `too_low_glucose(var(patient))`; the agent in charge of the patient informs the doctor (`msg_too_low_glucose(var(patient),var(doctor))`) and then it receives either a message from the doctor saying to continue to monitor, or a protocol switch request.
- `OtherSymptoms` sub-protocol: in case the patient has tremors or she is irritable (perception event types `irritability(var(patient))` and `tremors(var(patient))`), then a switch of the patient agent to the `severity1` protocol is required: the agent informs the doctor and then sends a switch request to itself; if the symptoms are convulsions, apnea, or irregular breathing, then the patient agents switches to the `severity2` protocol by sending the `switch_request(var(patient),var(patient),severity2)` message to itself.

```
trace_expr_template(standard_protocol, Standard) :-
StandardProtocol = Ok \/ TooLowGlucose \/ OtherSymptoms,
Ok = ok_glucose(var(patient)):
     msg_ok_glucose(var(patient),var(doctor)):StandardProtocol,
TooLowGlucose =  too_low_glucose(var(patient)):
    msg_too_low_glucose(var(patient),var(doctor)):
      (msg_continue_monitor(var(doctor),var(patient)):StandardProtocol)
        \/ switch_request(var(doctor),var(patient),severity2):lambda)),
SwitchToSeverity1 =
   msg_switched_to_severity1(var(patient),var(doctor)):
   switch_request(var(patient),var(patient),severity1):
   lambda,
LightSymptoms =
  ((irritability(var(patient)):lambda) \/ (tremors(var(patient)):lambda))
   * SwitchToSeverity1,
SevereSymptoms =
   ((convulsions(var(patient)):lambda) \/
   (apnea(var(patient)):lambda) \/
   (irregular_breathing(var(patient)):lambda)) *
   (switch_request(var(patient),var(patient),severity2):lambda),
OtherSymptoms = LightSymptoms \/ SevereSymptoms,
Standard = finite_composition(|,StandardProtocol,
                              [var(patient),var(doctor)]).
```

To make an example of how an event type can be defined, the `has_type` definition of `too_low_glucose(Patient)` is

```
has_type(percept(Patient,plasma_level_glucose(PlasmaLevel)),
too_low_glucose(Patient)) :-
```

```
hours_after_birth(Patient,HoursBirth),
((HoursBirth >= 1,  HoursBirth <= 2,  PlasmaLevel < 28);
 (HoursBirth >= 3,  HoursBirth <= 47, PlasmaLevel < 40);
 (HoursBirth >= 48, HoursBirth <= 72, PlasmaLevel < 48)).
```

which associates a perception of the glucose level in the plasma to the `too_low_glucose(Patient)` event type. The expected plasma levels given the hours from the baby's birth are based on medical literature.

For space constraints we do not show the `severity1` protocol. A fragment of the `severity2` protocol is shown below and the most relevant aspect is that, if during the execution of this protocol a low level of glucose is perceived, then a message is prompted on the screen associated with the patient, asking for an intravenous injection of glucose solution, and the intervention of the doctor is requested. Since the patient is following a `severity2` protocol, this means that she is in an almost critical situation. For this reason, even in case the glucose level is ok, the high-frequency and high-severity monitoring continues to go on (`NormalGlucose = ok_glucose(var(patient)):Severity2Protocol`). When the doctor will be confident enough that the most critical period has been overcome, she will explicitly send a protocol switch request to the patient, to roll back to the normal or lower severity protocol (not shown for space constraints).

```
trace_expr_template(severity2, T) :-
    Severity2Protocol=
       TooLowGlucose \/ NormalGlucose,
    NormalGlucose =
       ok_glucose(var(patient)):Severity2Protocol,
    TooLowGlucose =
       too_low_glucose(var(patient)):
       intravenous_inj_glucose_sol(var(patient), var(interface)):
       CheckGlucoseAfterInjection,
    ..........
```

Implementation in Jason

The doctors' and patients' agents driven by the protocol introduced in the previous section have been implemented on top of Jason. We run tests with up to 10 patients and 3 doctors, as shown in Figure 5.

Figure 6 shows a portion of a run involving only one doctor and one patient, to make it easier to follow the protocol evolution. The *patient1* agent has just received a protocol switch request from the *doctor1* agent, due to a low glucose level in the patient's blood; it cannot manage it immediately, so it saves the request. When it can implement the required protocol switch, *patient1* has to project the new protocol to follow (in this case the *severity 2* protocol) onto itself. The message "Instantiated protocol" is printed after the protocol projection and switch have completed. After that, *patient1* sends a message to the *screen1* agent, which simulates a user interface in this simplified scenario, that will print the message: "An intravenous injection of a 10% glucose solution is necessary"; this message will be read by the parents who will do the injection.

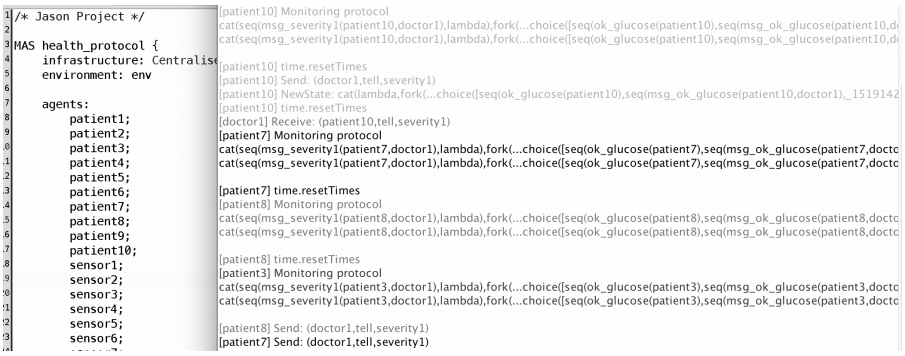


Fig. 5. Protocol execution in Jason: the MAS configuration is shown in the background.

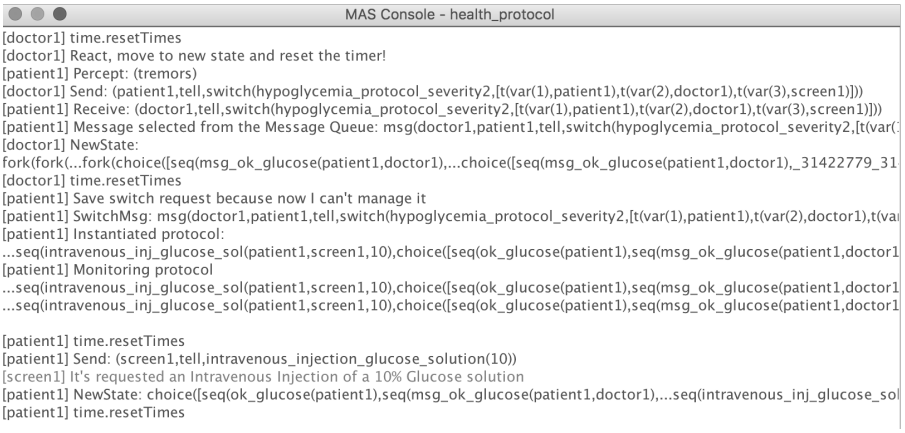


Fig. 6. Protocol execution in Jason: protocol switch request.

We simulated different situations where different sensory input was perceived by the patients' sensors, hence changing the course of actions and reactions, in order to test both our infrastructure and the protocols we have designed and implemented.

5 Conclusions and Future Work

Trace expressions are a compact and expressive formalism suitable for modeling protocols in multiagent systems and exploiting the protocol representation either for driving the agents behavior, or for verifying it at runtime, or both.

In [5] we have formally compared trace expressions with Linear Temporal Logic (LTL), a formalism widely adopted in runtime verification, and we have proved that for the purpose of runtime verification, trace expressions are strictly

more expressive than LTL: trace expressions are able to specify context-free and non context-free languages, while LTL is not.

In this paper we have presented an extension of our implemented framework for protocol-driven agents, consisting in the integration of trace expressions for protocol modeling and in the support to generic events in the protocol rather than just communicative events. The use of generic events instead of interactive actions only, allows developing more generic protocols which can model a wider range of situations.

The aim of this work was to show the potential of trace expressions for the specification of adaptive systems and the suitability of self-adaptive protocol-driven agents, which may decide to change their behavior at run-time based on the events they perceived, for safety-critical applications. In fact, all protocol-driven agents are coherent with the global protocol(s) which characterize the given domain and application, so their behavior is correct by construction. The lack of compliance to the protocol(s) may be due to events which are outside the agents control, namely events generated by the environment. Each agent must implement a policy for dealing with unexpected events. In the hypoglycemia scenario, the policy might be to warn immediately both the parents via the user interface, and the doctor, and move into an "alarm" state.

Although the protocols discussed in this paper are simple enough to be specified using regular languages, the ability of trace expressions to specify context-free and non context-free languages gives a great advantage over other widely used formalisms, in any application domain where a high modeling expressive power is required included the e-Health one.

The future developments of our work will be mainly devoted to study which kind of properties of a protocol expressed using the trace expression formalism can be verified statically. Static verification would be particularly relevant for scenarios in the e-Health domain, where reliability is a main goal to achieve.

References

1. D. Ancona, M. Barbieri, and V. Mascardi. Constrained global types for dynamic checking of protocol conformance in multi-agent systems. In *Proc. of SAC '13*, pages 1377–1379, 2013.
2. D. Ancona, D. Briola, A. Ferrando, and V. Mascardi. Global protocols as first class entities for self-adaptive agents. In *Proc. of AAMAS 2015*, pages 1019–1029, 2015.
3. D. Ancona, D. Briola, A. Ferrando, and V. Mascardi. Runtime verification of fail-uncontrolled and ambient intelligence systems: A uniform approach. *Intelligenza Artificiale*, 9(2):131–148, 2015.
4. D. Ancona, S. Drossopoulou, and V. Mascardi. Automatic generation of self-monitoring MASs from multiparty global session types in Jason. In *Proc. of DALT 2012*, volume 7784 of *LNAI*, pages 76–95. Springer, 2012.
5. D. Ancona, A. Ferrando, and V. Mascardi. Comparing trace expressions and linear temporal logic for runtime verification. In *Theory and Practice of Formal Methods*, volume 9660 of *LNCS*, pages 47–64. Springer, 2016.

6. R. Annicchiarico, U. Corts, and C. Urdiales. *Agent Technology and e-Health (Whitestein Series in Software Agent Technologies and Autonomic Computing)*. 1 edition, 2008.
7. E. A. Bayliss, J. F. Steiner, D. H. Fernald, L. A. Crane, and D. S. Main. Descriptions of barriers to self-care by persons with comorbid chronic diseases. *Annals of Family Medicine*, 1(1):15–21, 2003.
8. F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley, 2007.
9. F. Bergenti and A. Poggi. Developing smart emergency applications with multi-agent systems. *IJEHMC*, 1(4):1–13, 2010.
10. R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. John Wiley & Sons, 2007.
11. A. Bottrighi, F. Chesani, P. Mello, M. Montali, S. Montani, S. Storari, and P. Terenziani. Analysis of the GLARE and GPROVE approaches to clinical guidelines. In *Knowledge Representation for Health-Care: Data, Processes and Guidelines*, volume 5943 of *LNCS*, pages 76–87, 2009.
12. N. Bricon-Souf and C. R. Newman. Context awareness in health care: A review. *I. J. Medical Informatics*, 76(1):2–12, 2007.
13. V. Chan, P. Ray, and N. Parameswaran. Mobile e-health monitoring: an agent-based approach. *IET Communications*, 2:223–230(7), February 2008.
14. F. Chesani, P. D. Matteis, P. Mello, M. Montali, and S. Storari. A framework for defining and verifying clinical guidelines: A case study on cancer screening. In *Proc. of ISMIS 2006*, pages 338–343, 2006.
15. A. Ferrando. Parametric protocol-driven agents and their integration in JADE. In *Proc. of CILC 2015*, 2015.
16. M. Furmankiewicz, A. Sołtysik-Piorunkiewicz, and P. Ziuziański. Artificial intelligence and multi-agent software for e-health knowledge management system. *Informatyka Ekonomiczna – Business Informatics*, 2(32):51–62, 2014.
17. G. Ghinea, G. D. Magoulas, and A. O. Frank. Intelligent protocol adaptation for enhanced medical e-collaboration. In *Proc. of the Int. Florida Artificial Intelligence Research Society Conference 2003*, pages 276–280, 2003.
18. B. Horling and V. Lesser. A survey of Multi-Agent organizational paradigms. *The Knowledge Engineering Review*, 19(4):281–316, 2005.
19. D. Isern and A. Moreno. A systematic literature review of agents applied in healthcare. *J. Medical Systems*, 40(2):43:1–43:14, 2016.
20. S. Meystre. The current state of telemonitoring: a comment on the literature. *Telemedicine Journal & e-Health*, 11(1):63–69, 2005.
21. K. A. M. Nancy Wight. Abm clinical protocol #1: Guidelines for blood glucose monitoring and treatment of hypoglycemia in term and late-preterm neonates. *Breastfeeding Medicine*, 1(3), 2006.
22. I. Nunes, R. Choren, C. Nunes, B. Fábri, F. Silva, G. R. de Carvalho, and C. J. P. de Lucena. Supporting prenatal care in the public healthcare system in a newly industrialized country. In *Proc. of AAMAS 2010*, pages 1723–1730, 2010.
23. M. Schwaibold, M. Gmelin, G. von Wagner, J. Schöchlin, and A. Bolz. Key factors for personal health monitoring and diagnosis device. In *Mobile Computing in Medicine*, volume 15 of *LNI*, pages 143–150. GI, 2002.
24. E. M. Shakshuki and M. Reid. Multi-agent system applications in healthcare: Current technology and future roadmap. In *Proc. of ANT 2015*, volume 52 of *Procedia Computer Science*, pages 252–261. Elsevier, 2015.
25. B. Smith, D. M. Pisanelli, A. Gangemi, and M. Stefanelli. Clinical guidelines as plans - an ontological theory. In *Methods of Information in Medicine*, 2006.

Limitations and Divergences in Approaches for Agent-Oriented Modelling and Programming

Artur Freitas, Rafael C. Cardoso, Renata Vieira, and Rafael H. Bordini

Postgraduate Programme in Computer Science, School of Informatics (FACIN)
Pontifical Catholic University of Rio Grande do Sul (PUCRS). Porto Alegre, RS - Brazil
{artur.freitas, rafael.caue}@acad.pucrs.br,
{renata.vieira,rafael.bordini}@pucrs.br

Abstract. This paper shares our experiences in applying two well-known methodologies that play different roles in agent-oriented software engineering: modelling with Prometheus and programming with JaCaMo. First, we modelled a realistic multi-agent scenario using Prometheus to support its design and specification. Afterwards, JaCaMo was used as the development platform for programming our case study. Then, we were able to compare the outcome of combining these approaches, allowing us to identify some gaps, limitations and conceptual divergences when such approaches are used together to engineer a complex case study. This empirical study is our basis for discussing the lessons learned and for showing the theoretical and practical aspects of applying these two traditional agent-based approaches. Therefore, this paper highlights advantages and drawbacks of using Prometheus and JaCaMo to design and implement a complex multi-agent scenario, and how suitable is their integration for improving agent-oriented software engineering.

Keywords: Agent-oriented software engineering, Prometheus, JaCaMo

1 Introduction

Producing software code for complex and highly detailed systems directly in programming environments without first using any specification, modelling or design mechanism can cause many problems. For example, without a proper modelling of the system's environment it can be difficult to find potential bugs when they eventually appear in the implementation, since they might be bugs that were introduced during the design and specification of the system. These problems can cause further delay in the development of complex MAS, which are already notoriously hard to debug, mostly due to the lack of debugging support in MAS development platforms. Originally, methodologies for Agent-Oriented Software Engineering (AOSE) were aimed at agent-oriented programming languages that were mostly concerned with programming individual agents. Since then, programming abstractions covering the social and environmental dimensions of Multi-Agent Systems (MAS) have emerged [2], and are resulting in new MAS development platforms with multiple levels of abstractions. AOSE methodologies though, are not following the same pace. Although traditional methodologies can be used with these new MAS development platforms, they may diverge in

several points due to their differences regarding abstraction levels, and, thus, it is important to be aware if such combination can be successfully used, and which are the advantages, limitations and consequences of doing so. Therefore, our work discusses how these AOSE approaches can complement each other and which are the implications of such combination, rather than using each one of them in isolation.

Agent-oriented methodologies and programming languages for engineering MAS are often disjointed, resulting in limitations, gaps and conceptual divergences between the modelling and programming phases. In this paper, we apply two agent technologies in a complex case study in order to comparatively investigate the modelling and programming approaches for MAS. The case study allows us to point out and discuss the challenges we faced during this process, as well as possible ways to overcome them. Our case study is the Multi-Agent Programming Contest of 2016[1], an annual competition carried out as an attempt to stimulate research in the area of MAS development and programming. The performance of a particular system is determined through a series of simulation rounds, where systems compete against each other. This year the scenario consists of solving logistics problems in the realistic streets of cities, by buying, building, and delivering goods. First, Prometheus [6] is employed to specify our models, which are, then, used to code the system in the JaCaMo [1] programming framework. In other words, this work investigates the suitability of Prometheus for the specification of agent systems considering that the codification will take place in JaCaMo. In such context, it is important to be aware of divergences when these approaches are used in the development of complex MAS. Therefore, on the light of our case study, we analyse and discuss differences and problems (with possible solutions) between the combined use of the Prometheus methodology with the JaCaMo framework. As result, we point out lessons learned from using these technologies in combination.

This paper is structured as follows. Section 2 explains the basic concepts of the two agent-oriented modelling and programming approaches investigated in this paper: Prometheus and JaCaMo. Then, we describe the scenario and show our Prometheus models and pieces of JaCaMo code in Section 3, which is followed with a discussion about limitations, conceptual gaps, and possible solutions. Finally, Section 4 discusses related work, and our final remarks are given in Section 5, along with an outline of future research directions.

2 Background

The Prometheus methodology has been developed for over a number of years as a result of being used in the industry [3]. While Prometheus [6] is a methodology for modelling intelligent agent systems, the Prometheus Design Tool (PDT) is a graphical tool that follows the Prometheus methodology in order to build the design of MAS [8]. PDT started as a stand alone tool, but it is nowadays being developed as a plug-in for Eclipse. Prometheus contains three phases: *system specification*, *architectural design*, and *detailed design*. The *system specification* focuses on identifying basic system functionalities, along with inputs (percepts), outputs (actions), and any important shared

[1] https://multiagentcontest.org/

data sources. This phase defines what the system is intended to do. The *architectural design* uses the outputs from the previous phase to determine the system's agents and how they will interact. Thus, it establishes the structure of the system being developed. The *detailed design* looks at the internals of each agent and how it will accomplish its tasks within the overall system, that is, it establishes the plans that the agents require in order to achieve their goals.

JaCaMo [1] combines three separate technologies into a framework for MAS programming that makes use of multiple levels of abstractions, enabling the development of robust MAS. Each technology (Jason, CArtAgO, and Moise) was developed separately for a number of years and are fairly established on their own when dealing with their respective abstraction level. Jason is responsible for the agent level, it is an extension of the AgentSpeak language. Based on the BDI (Belief-Desire-Intention) model, agents in Jason react to events in the system by executing actions on the environment, according to the plans available in each agent's plan library. CArtAgO is based on the A&A (Agents and Artefacts) model, and deals with the environment level. Artefacts are used to represent the environment, storing information about the environment as observable properties and providing actions that can be executed through operations. Agents focused on artefacts can obtain percepts from them and execute operations on the artefacts. Moise handles the organisation level, enabling an explicit specification of the organisation in a MAS. This level adds new elements to the MAS, such as roles, groups, organisational goals, missions, and norms. Agents can adopt roles and compose groups. Missions are defined to achieve organisation goals, and the behaviour of the agents that adopt roles to execute these missions is guided by norms.

3 Modelling and Implementation of the Case Study

Our work uses PDT, a plugin for the Eclipse platform that adopts the graphical notation depicted in Fig. 1. This image identifies the symbols used in PDT's graphical models, serving as a notation for many of the models presented throughout this paper. Our case study is based on the multi-agent programming contest scenario of 2016. In this scenario two teams of autonomous vehicles are controlled by agents, competing against each other in order to accomplish logistic tasks in the streets of a realistic city. Completing tasks rewards the team with money, and the team with the most money by the end of the simulation is the winner. Tasks can be created by the environment or by one of the agent teams. A task can require the acquisition, assembling, and transportation of goods. In our simulations teams have four agents, with each agent having a different type. These types define the agent's speed, if they move by air or land, battery charge, maximum load, and what tools they can use. The types of agents are: car, drone, motorcycle, and truck. The map contains facilities of several types such as shops, warehouses, charging stations and storage facilities. Items can be bought, crafted, given to a teammate, stored, delivered as part of a job completion, recovered from a storage facility, and dumped. These action may only happen at their respective locations/facilities. Some overall characteristics assumed for our case study are given next. Each team is composed of only collaborative agents. The two teams are competing with each other to win money. The strategies for opponent teams are unknown (since we develop models

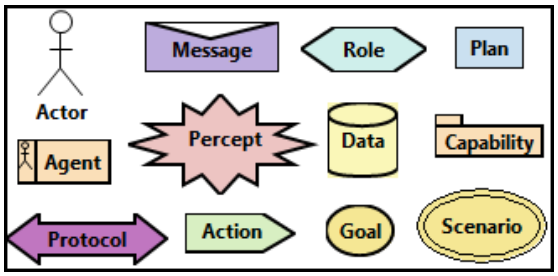


Fig. 1. Graphical notation of the Prometheus elements (obtained from the PDT in Eclipse)

and codes that correspond only to our team). It is assumed for our agents that they are truthful, their communication is reliable and their knowledge is imperfect.

3.1 System Specific Design

In Prometheus, the specification usually starts with the **analysis overview** diagram, which identifies `actors`, `percepts`, `actions`, and `scenarios` [6, 3]. `Actors` are external entities that will use or interact in some way with the system. Actors can be other software systems or humans. `Percepts` are the inputs to each `scenario`. `Actions` are produced by the system for each `scenario`. `Scenarios` describe the interaction and correspond to the main functionalities of the system. A `scenario` is a sequence of structured steps where each step can be one of: `percept`, `action`, `goal`, or (sub)`scenario`. `Goals` are defined as desires of `agents` and may trigger the execution of `plans`.

Figure 2 shows the analysis overview diagram. When mapping Prometheus to JaCaMo, Actors correspond to the artefacts of CArtAgO, and Actions can be seen as the operations of artefacts. The `round procedure` scenario, which represents each round of the simulation, models that agents interact with a `server` actor by receiving the `round start` and `round end` perceptions. A round consists of a series of steps in which agents receive the `request action` perception and then send an `action` to the actor. We used a contract net protocol mechanism, based on the original design of Smith [7], to distribute tasks among agents, as shown in the `task announcement procedure`, `bid procedure`, and `award procedure` scenarios in Fig. 2. The contract net was modelled using artefacts to control and mediate communication among agents by taking advantage of the shared resources available in CArtAgO, instead of using the usual method of message passing, and thus, improving performance during execution. Since agents have a deadline to send an action during each step, performance is an important feature in the strategy of any team. The `task board` is where tasks are announced. Each announced task has a respective `contract net board`, where agents make bids and the task is awarded to the best bid.

We observe the following limitations so far. In our case study, what is modelled as Actors in Prometheus is translated to Artefacts in JaCaMo. However, there is not any visual way to represent that an Actor, an Agent, or a Scenario instantiates an Actor. That is, none of these elements can connect directly with an Actor. In CArtAgO, artefacts can be created by: (i) agents; (ii) other artefacts; or (iii) when the MAS starts

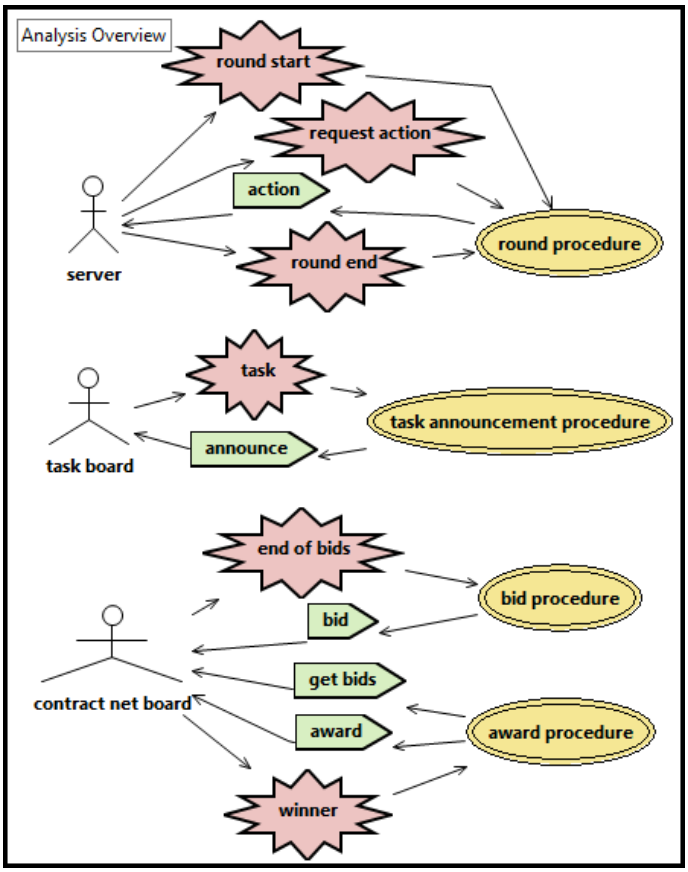


Fig. 2. Analysis overview diagram

its execution. We did not found any way to represent these things without having to change the meta-model of Prometheus and/or PDT. When using CArtAgO, both agents and artefacts can execute operations (i.e., actions). However, it is not possible to represent that an actor can execute an action in the Prometheus analysis overview diagram (e.g., a connection from an actor to an action). This diagram allows us only to represent that agents can execute actions, and that actors receive actions. To overcome this limitation, a scenario must be created between these elements, so that the actor connects with the scenario and the scenario connects with the action. Relations between two actors cannot be represented, thus, we cannot specify hierarchies among artefacts (e.g., to define inheritances/specialisations of artefacts/actors). The same limitation is true for hierarchies of agents, and hierarchies of roles. Another limitation is that agents cannot connect directly with actors, so there is no way of establishing that an agent can create an actor, or vice-versa. In CArtAgO, agents can only obtain perceptions and execute operations over artefacts after successfully focusing on them, and agents can only focus on artefacts if they are on the same workspace (localities where artefacts are situated). We did not found out how to represent these access restrictions.

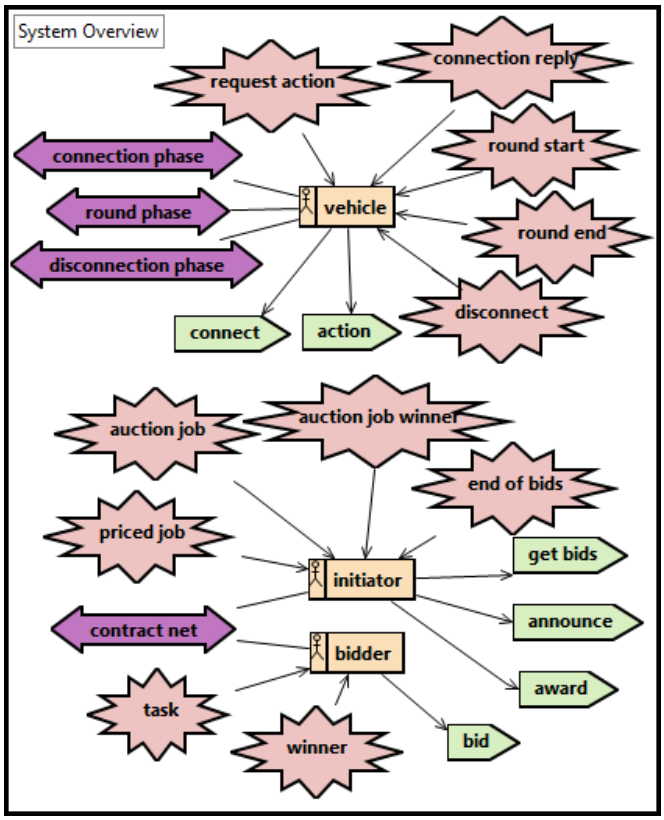


Fig. 3. System overview diagram

3.2 Architectural Design

The **system overview** diagram ties together the agents, events, and shared data objects [6]. `Agents` perform `actions`, receive `percepts`, bring together `roles`, exchange `messages`, and participate in `protocols`. `Protocols` define interactions between `agents` in terms of the allowable sequences of `messages` passed between them and the interactions with things outside the system (`actors`). `Messages` are sent and received by `agents` in order to accomplish the various aims of the system. `Roles` are intended as relatively small and easily specified chunks of `agent` functionality for grouping `goals` into cohesive units.

The system overview diagram, as shown in Fig. 3, provides a general understanding of how the system as a whole will function, and adds agents and protocols to the models. It also relates them with the previously described actions and percepts, and adds new ones if required. In our case, we have `vehicle`, `initiator` and `bidder` as agents. The elements introduced here are propagated and detailed better in further phases of Prometheus. Protocols and actors cannot be visually connected in any diagram (except in the sequence diagrams, as shown in Fig. 4 and Fig. 5). When we define that an agent participates in a protocol, this automatically propagates to the system overview diagram.

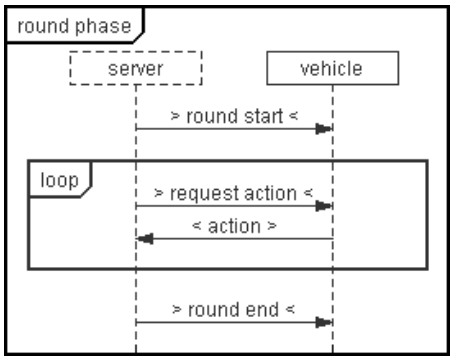


Fig. 4. Sequence diagrams of the `round phase` protocol

The same does not happen when it is defined that an actor participates in a protocol (for example, such relation does not propagate to the `analysis overview` diagram). It is not possible to define what triggers or initialises a protocol, as this information cannot be included in any diagram. The very first thing in the protocol indicates that it has begun, but it does not indicate the reason that made it start.

Percepts, actions, and actors can participate in protocol specifications, in addition to messages and agents. In protocols, percepts (represented by >perception_name<) originate from actors (represented by dashed rectangles) and go to agents (represented by solid rectangles); and actions (represented by <action_name>) always originate from an agent and go to an actor. Figure 4 depicts the sequence diagram for the `round phase` protocol and Fig. 5 illustrates the `contract net` protocol. Our contract net protocol starts when an `initiator` agent performs an `announce` action in a `task board` artefact. Then, the `task board` creates a `contract net board` artefact, however Prometheus and PDT do not allow for such information to be included in the diagram, and thus, it is not visually represented. Then, the `contract net board` produces a `task` percept for `bidders` that will `bid` for it. The `contract net board` controls the `deadline` and sends the `end of bids` percept to the `initiator`, which will `get bids` and `award` the best offer (`deadline` is achieved when all agents that are able to bid do so, or when a timeout is reached). In the end, the `bidder` receives a `winner` percept indicating who won the task.

As we previously commented, it is not possible to connect two actors in the sequence diagram. This is needed to define, for example, that an actor creates another actor. In our case, `task boards` make instances of `contract net boards` when an `initiator` announces a task. We already discussed the representation of actors' instantiations when analysing limitations in the analysis overview diagram, and the same observations apply to the sequence diagrams. Figure 6 shows (some parts of) the `contract net board` CArtAgO artefact code. We made adaptations from the contract net protocol artefacts that come with the CArtAgO package in order to use it in our implementation. This artefact corresponds to the `actor` represented in Prometheus using the same nomenclature. Its operations (`bid` and `getBids`) are traced to the `actions` in Prometheus that previously appeared in several diagrams (Figures 2, 3, and 5). The `bid` operation adds the bid received as parameter in a bid list if the state

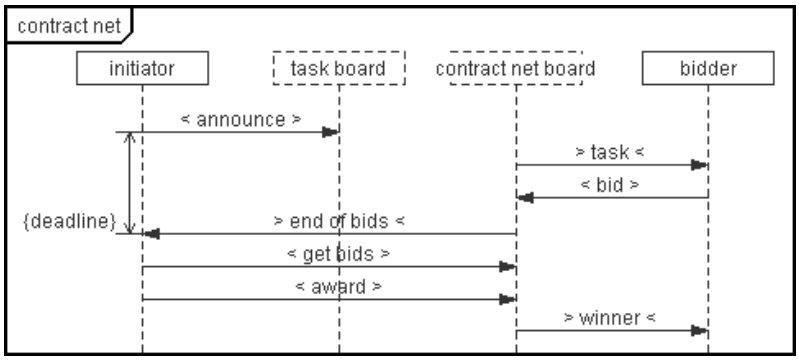


Fig. 5. Sequence diagrams of the `contract net` protocol

of this `contract net board` is `open`. The `getBids` operation returns the bid list after passing the guard condition that checks if the bidding window is `closed`. Moreover, the observable properties of this CArtAgO artefact (defined in the `init` method) are related to the `percepts` of Prometheus' models. Some parts of the code are omitted for simplicity and due to the lack of space. One example is the Bid class (appears on lines 2, 13, 22), which is used to manage the bids and contains as attributes an identification number and a value.

Our **data coupling overview** diagram is depicted in Figure 7. In Prometheus, the data coupling diagram contains the `roles` (functionalities) and their relations to all identified `data` (not only persistent data, but also data required by the functionalities). Therefore, in our case study, we found that the data can come from two different sources: (i) from the environment (more specifically, the artefacts); or (ii) in the form of beliefs. In both cases, we found that, in this context, Prometheus presented conceptual divergences regarding the data representation adopted in JaCaMo.

3.3 Detailed Design

The detailed design phase consists of a list of agent overview diagrams, one for each agent has their own capability overview diagrams, one for each capability included in the agent [3]. The **agent overview** diagram provides the top level view of the agent internals [6]. It is similar in style to the system overview diagram, but instead of agents within a system, it shows capabilities within an agent. Then, the **capability overview** diagram goes even further, describing the internals of a single capability. At the bottom level these will contain plans, with events providing the connections between plans, just as they do between capabilities and agents. These diagrams are similar in style to the system and agent overview diagrams, although plans are constrained to have a single incoming (triggering) event.

`Capabilities` of `agents` are defined in terms of `plans`, `events`, and `data` [6]. `Capabilities` are described by a capability descriptor which contains information about its external interface – the events that are used as inputs, and the events that are produced as inputs. The `capabilities` of an `agent` usually (at least initially) correspond to the `roles` that were assigned to it, though `roles` may also be split into

```
public class ContractNetBoard extends Artifact {
    private List<Bid> bids = new ArrayList<Bid>();
    private int bidId = 0;

    void init(String taskDescr, long duration){
        defineObsProperty("task_description", taskDescr);
        defineObsProperty("deadline", duration);
        defineObsProperty("state","open");
        // ...
    }
    @OPERATION void bid(int bid, OpFeedbackParam<Integer> id){
        if (getObsProperty("state").stringValue().equals("open")){
            bids.add(new Bid(++bidId,bid));
            id.set(bidId);
        } else
            failed("cnp_closed");
    }
    @OPERATION void getBids(OpFeedbackParam<Literal[]> bidList){
        await("biddingClosed");
        int i = 0;
        Literal[] aux = new Literal[bids.size()];
        for (Bid p: bids)
            aux[i++] = p.parseBid();
        bidList.set(aux);
    }
    // ...
}
```

Fig. 6. CArtAgO code for the `contract net board` artefact

multiple smaller `capabilities` or merged into a larger one. `Plans` are procedures that can be triggered by events such as the desire to achieve a `goal` or the belief of perceiving something. The concept of `Data` in Prometheus allows representation of domain information and entities that are outside of the agent paradigm [3].

Figure 8 shows a snippet of the agent overview diagram for `initiator` agents. This diagram shows the following 5 capabilities: *priced job analysis*, *auction job analysis*, *auction job winner verification*, *announcement procedure*, and *award procedure*. The `priced` or `auction job` percept triggers its corresponding `analysis` capability, which uses as data `map information` and/or `vehicle information`. When our `initiator` decides that a priced job will be taken, it creates the goal of `separate tasks`, which will be the input of the `announcement procedure`. When it is decided that an auction job is worth to take, the path is a little longer, because we have to make a bid to the server, and our team may not be the bid winner. So, we make a bid and save it, until we get a percept from the server of who is the `auction job winner`. If we verify that we won, then we can add the goal of `separate tasks`. This goal starts the `announcement procedure`, which divulges the task to our bidder agents. The next and final step is the `award procedure`, which collects the bids and `award` one of our agent to execute the desired task.

Figure 9 shows a snippet of the agent overview diagram for `bidder` agents, focusing on the `bid procedure` and its corresponding capability overview diagram. The `bid procedure` is responsible to analyse the context and calculate a bid for a task. Thus, it receives as input the percept of a `task`, and data about the map and vehicles;

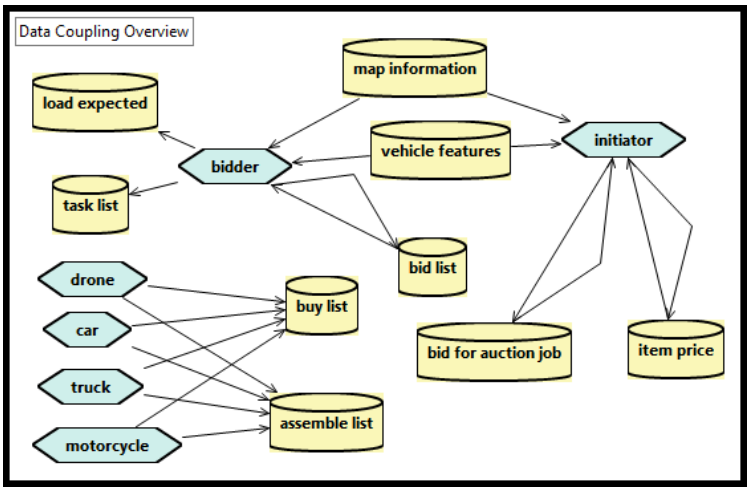


Fig. 7. Data coupling overview diagram

and produces as output the `bid` action, storing its value as a belief. The `focus` action happens on a `contract net board` artefact, which receives the `bid` action. Among the `vehicle features` data, it is the vehicle speed, load capacity, battery, tools, and so on. Figure 9 also shows code in JaCaMo which corresponds to the capability overview diagram of `bid procedure`. It shows two plans in JaCaMo (`+task` and `+!calculate_bid`) that correspond to the two plans in PDT. The icons of plans in PDT are directly converted to plans in Jason with their corresponding triggering events and plan bodies composed of actions, goals and data manipulations.

Figure 10 illustrates the capabilities and plans of vehicles agents, such as, for example, the `find items` and `get items` capabilities. The plan to `find items` build a list of items to buy and a list of items to assemble. Then, the plan to `get items` uses such lists as inputs to generate goals corresponding to the actions of `goto`, `buy` and `assemble`. Figure 10 shows Jason code for the `vehicle` agent, with the `plans` to achieve the `goals` of `find items` and `get items`. These code is traced to the elements represented in the Prometheus diagram illustrated in Figure 10. The `find items` plan has queries to discover which products must be bought and which are the best shops and workshops for that vehicle to acquire its desired resources. Some items cannot be bought directly in shops, but can only be obtained from assembling some specific items in a workshop. Thus, the `find items` creates beliefs about which items should be bought in each shop, and which workshops should be visited in order to assemble the composite items. The plan to `get items` creates the goals of `go to` each of these locations, `buy` the items and make the desired `assembles`.

Visually, the diagrams lack some graphical notion of ordering among the elements in a plan in the agent overview and capability overview diagrams. Plans are usually defined as an orderly or step-by-step conception or proposal for accomplishing an objective. A visual notation that could represent such things would be very interesting for modelling plans of MAS in the diagrams of Prometheus. In Moise, plans are composed as a list of sub-goals and an operator that specifies *sequence*, *choice* or *parallelism*.

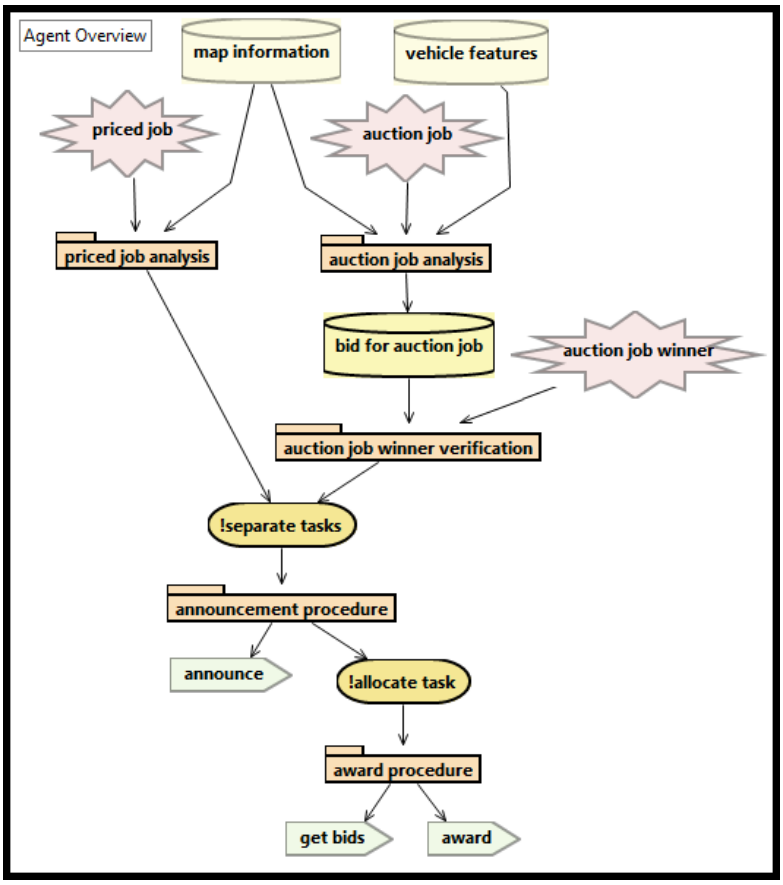


Fig. 8. Agent overview diagram for `initiator` agents

Thus, there is a mismatch between Prometheus and Moise where information would be lost or missing when converting from one specification to another.

We found confusing the fact that it is possible to place plans in the agent overview diagram. Prometheus works with the concept of capability, and each capability must be implemented through at least one plan in its corresponding capability overview diagram. Then, if a plan is placed in the agent overview diagram, it does not fit neither contribute to any capability that the agent is supposed to have.

Some aspects that did not contribute significantly during the modelling or development of our case study are now discussed. The **scenario overview** diagram shows only one kind of element (scenarios) and they cannot be linked, which is not very useful from the viewpoint of graphical models. When clicking on a scenario, it is possible to define its properties in textual formats and the steps of the scenario in tables (not very visually friendly formats). Also, it is not possible to define properties or additional information in relationships among PDT elements. For example, consider that we could link roles, then we could say that they are disjoint, or that a role specialises another, and

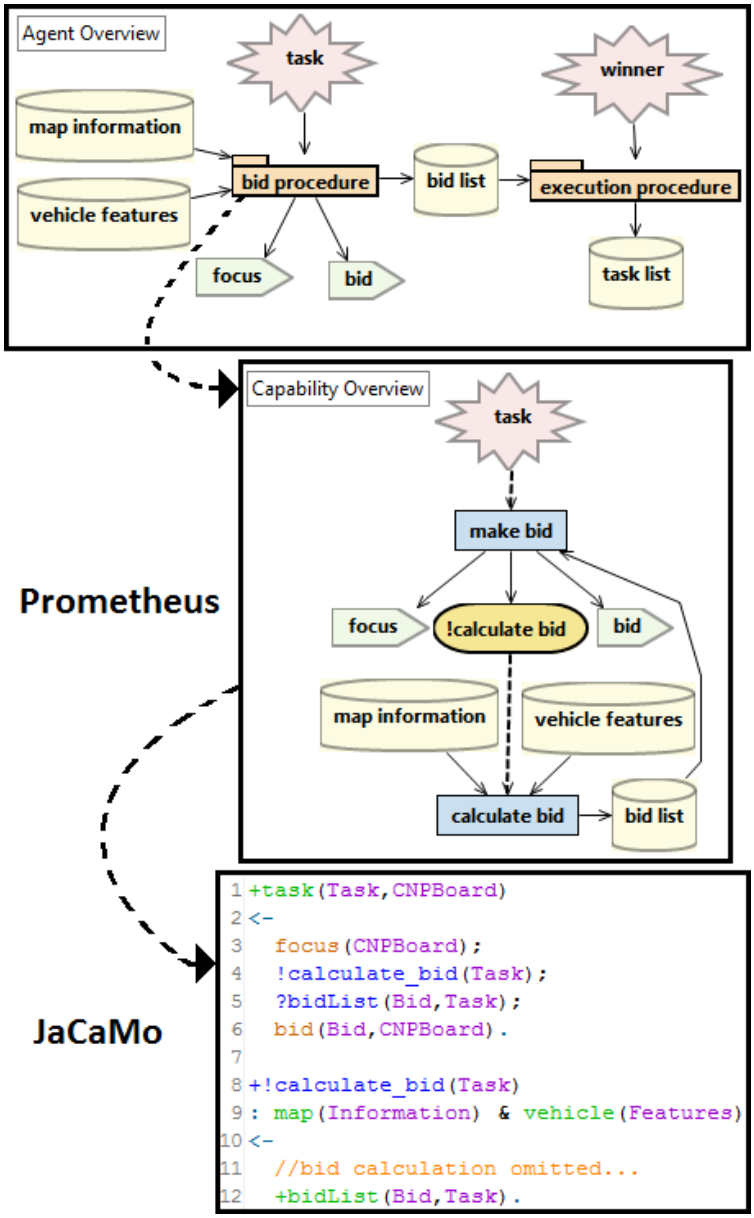
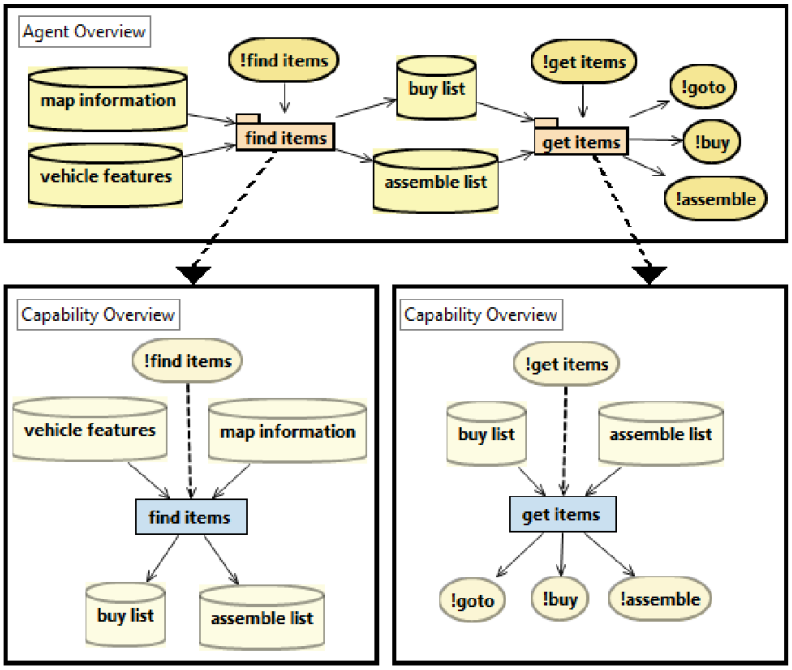


Fig. 9. Snippets of the agent overview diagram for `bidder` agents, capability overview diagram for `bid procedure`, and corresponding JaCaMo code (more specifically, Jason code)




```
1+!find_items
2     : working(JobId,ItemList,StorageId)
3<-
4     for ( .member(item(ItemId,Qty),ItemList) ) {
5         ?product(ItemId,Volume,BaseList);
6         if (BaseList == []) {
7             ?find_best_shop(ItemId,Shop);
8             +buyList(ItemId,Qty,Shop);
9         }   else {
10            ?decomp(BaseList,BaseItems);
11            for ( .member(item(BaseItemId,BaseQty),BaseItems) ) {
12                ?find_best_shop(BaseItemId,Shop);
13                +buyList(BaseItemId,BaseQty,Shop);
14            }
15            ?find_best_workshop(Workshop);
16            for ( .range(I, 1, Qty) ) {
17                +assembleList(ItemId,Workshop);
18            }
19        }
20    }.
21
22+!get_items
23<-
24    for ( buyList(_,_,Shop) ) {
25        !goto(Shop);
26        while ( buyList(Item,Qty,Shop) ) {
27            !buy(Item,Qty);
28        }
29    }
30    while ( assembleList(Component,Workshop) ) {
31        !goto(Workshop);
32        !assemble(Component);
33    }.
```

Fig. 10. Capabilities and plans of `vehicle` agents in Prometheus and corresponding Jason code

so on for other types of relationships. According to the papers of Prometheus [3], the steps of scenarios can assume only one of the following types: percept, action, goal or (sub)scenario. However, PDT allows roles to be steps of scenarios, and the meaning for this is not clear for us. We think that protocols should be able to be added as steps of a scenario and, currently, PDT does not allow for such thing.

The **goal overview** diagram addresses only goals. Since PDT creates a goal for each scenario, and the scenarios correspond to the main functionalities of the system [3], the goals are best viewed as system's use cases. However, later diagrams treats goals as triggering of plans, and then goals are better interpreted as individual desires of agents. We believe that these different views on goals can be confusing and their meaning should be defined more precisely. The BDIMessage addresses only goals, however messages could also transmit beliefs and plans (when considering JaCaMo). Some other diagrams, such as **system role**, **data coupling** and **agent role grouping overview**, did not add much information for our case study.

Plans can have only one triggering condition, and there is not a direct and simple graphical representation of the context for plans (i.e., predicates that must be true to consider a given plan applicable). Also, it would be interesting to represent conditions (semantically different from the plan trigger) that can lock or unlock the execution of a plan, such as the presence of a belief resulted from a percept of something (e.g., vehicles can have a plan that waits until a percept of request action comes from the server). Two entities cannot have the same name [3], for example a goal and a plan. This impossibility can result in more advantages than limitations, but in our case it was the opposite. For example, a goal can be the triggering condition of a plan in Jason, so we would like to use to same name to refer to these two different (but related) entities. Also, it is not possible to have references to non-existent entities, since creating a reference will create the entity if it does not exist and when an entity is deleted all references to it are deleted as well. The propagation of changes is, usually, a good thing, but can generate unexpected results when we would like to change something only in one place, or we are not aware about other changes that resulted from a single modification.

4 Related Work

Prometheus and PDT are used to develop the design of a conference management system case study [5]. This work pointed out the importance of integrating Prometheus with other agent software design tools and methodologies. Then, the conference management system case study was also modelled with O-MaSE and Tropos in another paper [3], which compared Prometheus with its alternatives based on such example. We differ from these papers since, besides modelling a more complex scenario, we focus on pointing out limitations and divergences of Prometheus with regards to JaCaMo. Thus, rather than showing cases that could be successfully modelled, this paper highlights and discusses situations that were impossible to model or that turn out to be more confusing than enlightening.

Prometheus AEOlus [9] allows the integrated development of the three MAS dimensions (agent, environment and organisation) which contributes with: *(i)* a new meta-model that combines the meta-models of Prometheus and JaCaMo; *(ii)* a new interactive

incremental process based on the Prometheus process; and *(iii)* a code generation approach for JaCaMo based on this new meta-model. Prometheus AEOlus improves modelling, code generation and reduces the conceptual gap between the analysis and implementation phases. It extends Prometheus to include concepts that improve the modelling and code generation of the environment and organisation dimension for JaCaMo programming platform, where JaCaMo concepts were used to improve Prometheus development process to ensure that concepts used during the design and analysis stages will be used in the implementation stage. The code generation in Prometheus AEOlus requires the refinement of entities in the model to generate code (for JaCaMo components, i.e., Jason, Moise and CArtAgO). Thus, the models must be refined to include platform-specific information, and once the first version of code is generated, the models are no longer used during the programming step to complete the MAS development.

Research in the direction of tools for developing MAS through exploiting model-driven engineering techniques have led to a new proposal [4] of using Ecore with Prometheus. Ecore is used by the Eclipse Meta-modelling Framework to define meta-models, and it is applied to develop the meta-model concepts specific to Prometheus. More specifically, it addressed the generation of MAS graphical editors based on the models and how agent code generators can be developed from such visual models. In the end, MAS programming code can be automatically generated from the models, ranging from code skeletons to completely deployable products. To demonstrate this claim, templates have been created to automatically generate code in JACK language. Once the model is converted to code, the developer must continue the programming without using the model.

5 Final Remarks

This paper uses a MAS case study to highlight a number of discrepancies between the paradigms behind methodologies for AOSE and programming languages for MAS. In particular, the paper focuses on Prometheus [6] as the agent-oriented methodology and JaCaMo [1] as the framework for multi-agent programming. On one hand, Prometheus is one of the most well-known MAS model and methodology for developing intelligent agent systems. On the other hand, JaCaMo is a framework for MAS programming that combines three separate technologies: Jason for coding autonomous agents in AgentSpeak, CArtAgO for programming the environment as artefacts in Java, and Moise for specifying MAS organisations in XML. This paper highlights some aspects of MAS not covered or not aligned by models in Prometheus [6] when JaCaMo [1] is considered as the coding platform. The identification of mismatches between software engineering toots and programming languages can help in improving both and could result in a better alignment between them. Even for the concepts referred by the same name, there are differences and mismatches in the precise meaning of them. In fact, we provide evidences of gaps between the investigated approaches which allows to derive some important conclusions. For example, while many methodologies were proposed for agent programming in the past, they are not sufficient for the new and emerging techniques in agent programming, such as dealing with the multiple abstraction levels and not focusing only on the agents as individuals.

Our analysis of these techniques in practice allowed us to identify points for improvements. Such empirical study evaluates the state of the art in technologies and guide the development of new technologies based on the limitations of current ones. An interesting continuation of our work is to explore the same case study using alternatives for Prometheus [6] and JaCaMo [1]. We plan on expanding our models with more detailed strategies in Prometheus for this case study. Based on that, we want to advance our explorations on the relations of Prometheus with JaCaMo. The findings described in this paper are supported by a single case study, so we have to be careful to expand or generalise such discoveries for different contexts. Eventually, after trying other case studies and confirming the limitations shown here it will be possible to propose new approaches or enhancements for such AOSE approaches. Some limitations we faced when using Prometheus were already pointed out by their authors and claimed as future work [6], such as, for example, the introduction of social concepts to improve the models. However, these improvements are not available in the latest official version of PDT. One could question some of our modelling decisions, however, this paper is less about the details of the case study, and more about the broader vision towards AOSE approaches. Another interesting discussion is the relation of how similar problems exist and/or have been solved over the past 20 years in the general Software Engineering (SE) literature. A clear understanding of the evolution in mainstream SE can be crucial for evolving AOSE in the right direction.

References

1. Boissier, O., Bordini, R.H., Hübner, J., Ricci, A., Santi, A.: Multi-agent oriented programming with JaCaMo. Science of Computer Programming 78(6), 747–761 (2013)
2. Bordini, R.H., Dix, J.: Programming multiagent systems. In: Weiss, G. (ed.) Multiagent Systems 2nd Edition, chap. 11, pp. 587–639. MIT Press (2013)
3. DeLoach, S.A., Padgham, L., Perini, A., Susi, A., Thangarajah, J.: Using three AOSE toolkits to develop a sample design. International Journal of Agent-Oriented Software Engineering 3(4), 416–476 (2009)
4. Gascueña, J.M., Navarro, E., Fernández-Caballero, A.: Model-driven engineering techniques for the development of multi-agent systems. Engineering Applications of Artificial Intelligence 25(1), 159–173 (2012)
5. Padgham, L., Thangarajah, J., Winikoff, M.: The prometheus design tool a conference management system case study. In: Agent-Oriented Software Engineering VIII, LNCS, vol. 4951, pp. 197–211. Springer Berlin Heidelberg (2008)
6. Padgham, L., Winikoff, M.: Prometheus: A methodology for developing intelligent agents. In: Agent-Oriented Software Engineering III. LNCS, vol. 2585, pp. 174–185 (2003)
7. Smith, R.G.: The contract net protocol: High-level communication and control in a distributed problem solver. IEEE Trans. Comput. 29(12), 1104–1113 (Dec 1980)
8. Sun, H., Thangarajah, J., Padgham, L.: Eclipse-based prometheus design tool. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems. vol. 1, pp. 1769–1770. IFAAMAS (2010)
9. Uez, D.M., Hübner, J.F.: Environments and organizations in multi-agent systems: From modelling to code. In: 2nd International Workshop on Engineering Multi-Agent Systems. pp. 181–203 (2014)

Application Framework with Abstractions for Protocol and Agent Role

Bent Bruun Kristensen

Maersk Mc-Kinney Moller Institute, University of Southern Denmark, Denmark
bbkristensen@mmmi.sdu.dk

Abstract. In multi-agent systems, agents interact by sending and receiving messages and the actual sequences of message form interaction structures between agents. Protocols and agents organized internally by agent roles support these interaction structures. Description and use of protocols based on agent roles are supported by a simple and expressive application framework.

Keywords: Multi-agent system, Protocol, Agent role, Reactive and proactive role, Application framework.

1 Introduction

Agents are active, autonomous, and smart, i.e. among others capable of reactive and pro-active behavior [1]. A multi-agent system consists of a number of agents interacting with one-another—and to successfully interact, agents require the ability to cooperate, coordinate, and negotiate with each other. We describe interactions by protocols that relate agent roles of such communicating agents.

Abstraction is essential: "Without abstraction we only know that everything is different" [2] meaning that use of abstractions to describe observations is essential for the resulting understanding. Dialogues and agent roles are abstractions, i.e. by these concepts the developer can understand and describe the organization structure of agents as well as the interaction structure between communicating agents.

Our aim is to support protocols that capture communication between agent roles by an object-oriented application framework [3] with agents, reactive role, proactive role and message. The underlying agent model and the application framework are illustrated by the Contract Net [1]. The model and framework are compared to related work and evaluated.

2 Agent Model

Reactive and Proactive Roles. Agents communicate by sending and receiving messages representing events as illustrated in Fig. 1. An agent consists of a varying number of reactive and proactive roles. Reactive and proactive roles are abstractions for internal organization of an agent and messages are sent from and received by these

roles. If a message is sent to the agent itself a default reactive role of the agent receives the message.

The roles of an agent execute one at a time and in a non preemptive way [4], i.e. they exhibit cooperative multitasking, in which case a role can self-interrupt and voluntarily give up control. Reactive and proactive roles are stereotypes but combinations can be described. Each reactive and proactive role has a list of messages to be handled on a first come first served basis. A reactive role repeats the execution of an action to take care of its list of messages whenever the handling of the previous message is completed and the awaiting message list is not empty. A proactive role consists of a single execution of an action that takes care of pausing as well as waiting and handling messages until its purpose is completed.

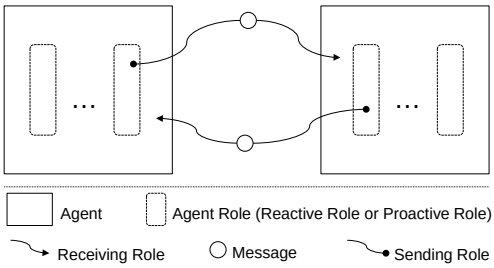


Fig. 1. Agents organized by reactive and proactive roles.

Protocols. A protocol describes a process where an initiator initializes the interaction by sending messages to a number of participants where after these participants may reply to the initiator as part of the interaction, etc. The protocol takes place between proactive roles of agents. The protocol and the proactive roles together form abstractions over an interaction structure between the involved agents.

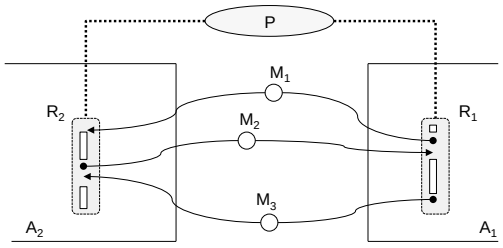


Fig. 2. Illustration of protocol and proactive roles

Fig. 2 illustrates a protocol P between proactive roles R_1 and R_2 in agents A_1 and A_2. Role R_1 initializes the interaction by sending a message M_1 to role R_2. Role R_2 replies with message M_2 to R_1. In this manner a protocol between R_1 and R_2 may describe a continued interaction between R_1 (from A_1) and R_2 (from A_2). The protocol illustrated in Fig. 2 is similar to the coroutine mechanism of SIMULA [5] in the sense that a role sends a message, immediately suspends itself and the receiving role is resumed.

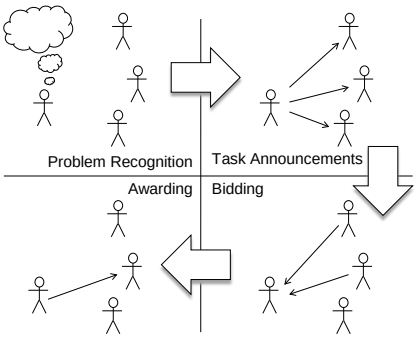


Fig. 3. Illustration of Contract Net

Example: Contract Net. Fig. 3 illustrates the Contract Net with a collection of stickmen. Each stickman in the collection can, at different times or for different tasks, be involved in several simultaneous tasks as both manager and contractor. When a stickman gets a composite task (or for any reason cannot solve its present task), it breaks the task into subtasks (if possible) and announces them (acting as a manager), receives bids from potential contractors, and then possibly awards a contractor. If no bids are received after a given period of time the manager gives up the negotiation. If a bid is not awarded after a given period the contractor gives up the negotiation.

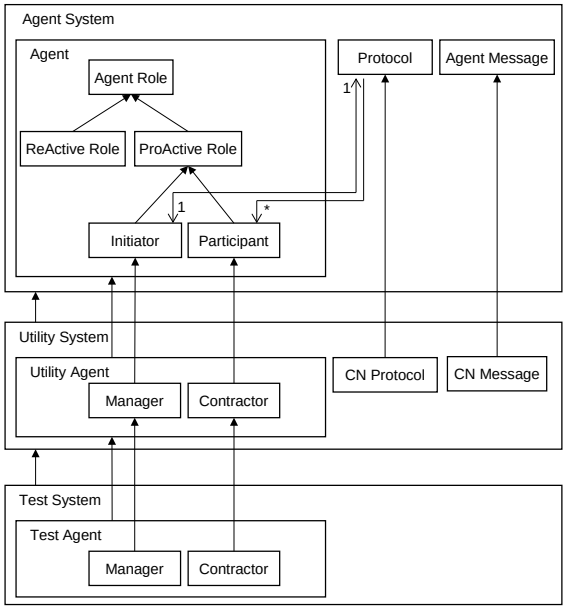


Fig. 4. Conceptual Model showing the contents of and relations between `Agent System`, `Utility System` and `Test System`

A model for the Contract Net includes: A protocol is set up with a proactive role for the manager agent (the initiator) and a proactive role for each of the contractor agents (the participants). A manager maintains a negotiation by initiating an interaction with a number of contractors. A contractor receives a task announcement and may reply with a bid to the manager. Having received bids the manager chooses among these and may reply with an award to the chosen contractor in which case a contract is established.

3 Framework Overview

Fig. 4 illustrates the conceptual model of the application framework with `Agent_System` that is specialized to another application framework `Utility_System` (to support various protocols with the Contract Net as an example) that in turn is used in the application `Test_System`. The contents of and relations between `Agent_System`, `Utility_System` and `Test_System` are described in the following sections.

```
… class Agent_System {

  … abstract class Protocol {
    … Protocol (Agent initiator, Agent[] participant) {
      …
      initiatorRole = initiator.newInitiatorRole(this);
      for (int i = 0; i< …; i++) {
        participantRole[i] =
                    participant[i].newParticipantRole();
      };
    }

    …
    … Agent.Initiator initiatorRole;
    … Agent.Participant[] participantRole = … ;
  }

  … abstract class Agent extends … implements … {
    …
    … abstract class Agent_Role extends … {…}

    … abstract class ReActive_Role extends Agent_Role {
      abstract … int Act(Agent_Message am)
      …
    };

    … abstract class ProActive_Role extends Agent_Role {
      abstract … void Act()
      …
    }

    … abstract class Initiator extends ProActive_Role {…}
    … abstract class Participant extends ProActive_Role {…}
    … abstract Initiator newInitiatorRole(Protocol p)
    … abstract Participant newParticipantRole();
  }

  … abstract class Agent_Message extends EventObject {…}
}
```

Fig. 5. Application Framework: `Agent_System`

4 Application Framework: Agent System

Fig. 5, 6 and 7 show extracts of the textual version of the application framework `Agent_System` with classes and methods shown in grey. Fig. 5 shows class `Agent_System` with abstract classes `Agent`, `Protocol` and `Agent_Message`. Class `Agent` has abstract classes `ReActive_Role` and `ProActive_Role` (extending class `Agent_Role`). Class `ProActive_Role` is extended to classes `Initiator` and `Participant` both related to class `Protocol`. In addition classes `ReActive_Role` and `ProActive_Role` include the abstract method `Act(…)`. The `Act(…)` method of class `ReActive_Role` returns a delay until next invocation and is invoked repeatedly with the next message as parameter while messages are waiting. The `Act()` method of class `ProActive_Role` is invoked only once and its execution may include pausing, awaiting messages, etc. until its execution is completed. Abstract methods `newInitiatorRole` and `newParticipantRole` are used by `Protocol` to instruct actual specializations of `Agent` to instantiate actual specializations of `Initiator` and `Participant`. Class `Protocol` instantiates and starts the execution of `InitiatorRole` (with the `Protocol` object as parameter) for `InitiatorAgent` and of `ParticipantRole` for each of the `ParticipantAgents`.

```
… abstract class Agent_Role extends … {
  … void rolePause(int sleepTime) {…}
  … Agent_Message roleAwait() {…}
  … void replyMessage(Agent_Message rm, Agent_Message am) {…}
  … Agent_Message handleMessage() {…}
  …
}
```

Fig. 6. Interaction methods of class `Agent_Role`

Fig. 6 shows extracts of selected interaction methods of class `Agent_Role` (inherited by `ReActive_Role` and `ProActive_Role`):

- `rolePause(…)`, the role pauses for a period of time
- `roleAwait()`, the role waits until a message is received and then returns the message
- `replyMessage(…)`, a message is sent to a role of another agent as a reply to a message received from that role
- `handleMessage()`, the next waiting message is returned (if any and else `null`)

```
… abstract class Agent_Role extends … {
  … void initiateProtocol(Agent_Message am, Participant p) {…}
  … void replyProtocol(Agent_Message ram, Agent_Message am) {…}
  …
}
```

Fig. 7. Methods `initiateProtocol` and `replyProtocol`

Class `Protocol` sets up the protocol between agent roles—the `Initiator` and the `Participants` agents. Fig. 7 shows extracts of the interaction methods related to the protocol:

- `initiateProtocol(…)`, the `Initiator` sends a message `am` to a `Participant` to initialize the protocol.
- `replyProtocol(…)`, either the `Initiator` or a `Participant` send a message `ram` in reply to message `am` received within the protocol.

5 Application Framework: Utility System

Class `Agent_System` can be used directly to construct a multi-agent system with reactive and proactive agent roles and protocols. We choose to extend `Agent_System` to another abstract class `Utility_System` to illustrate an example of an abstract protocol—the Contract Net. Classes `Utility_System` is then specialized in class `Test_System` as an actual use.

Fig. 8 shows the ingredients of the specialization of `Agent_System` to `Utility_System`: `Protocol` is specialized to `CN_Protocol` and the proactive roles `Initiator` and `Participant` to `Manager` and `Contractor`, respectively. Classes `Manager` and `Contractor` specify their own `Act()` method according to the Contract Net. And each of these `Act()` methods makes use of additional methods (shown as dotted) to be implemented in the actual use of the `Utility_System`.

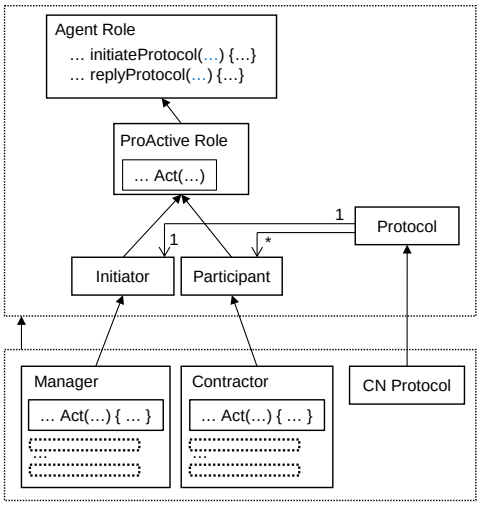


Fig. 8. Specialization of `Agent_System` to `Utility_System`

Fig. 9 illustrates how `Utility_System` and `Utility_Agent` extend `Agent_System` and `Agent`, respectively. Class `Protocol` is extended to `CN_Protocol` that initializes a `Manager` role for the `Initiator` agent and a `Contractor` role for each of the `Participant` agents.

```
… abstract class Utility_System extends Agent_System {

  … class CN_Protocol extends Protocol {
    … CN_Protocol (Agent initiator, Agent[] participant) {
      super(initiator, participant);
      …
    }

    … Utility_Agent.Manager managerRole;
    … Utility_Agent.Contractor[] contractorRole = … ;
    …
  }

  … abstract class Utility_Agent extends Agent  {

    … abstract class Manager extends Initiator {
      … Manager (Protocol protocol)  {…}
      … void Act() {…}
    };

    … abstract class Contractor extends Participant {
      … void Act() {…}
    }
  }
}
```

Fig.9. `Utility_System` with `CN_Protocol`

When a `Protocol` is instantiated as shown in Fig. 10 its constructor initializes `ManagerRole` through the executing agent and `ContractorRoles` through `otherAgents`.

```
new CN_Protocol(Test_Agent.this, otherAgents);
```

Fig. 10: Creation of a `CN_Protocol`

Fig. 11 shows abstract class `Manager` as an extension of `Initiator`. Method `Act()` of class `Manager` uses the abstract methods (*hot spots* cf. [6]) in italics (implemented in `Test_System`).

```
… abstract class Manager extends Initiator {
  …
  … void Act() {…}

  … abstract CN_Task createCN_Task(int t)
  … abstract CN_Task.Offer createOffer(CN_Task t)
  … abstract CN_Message newOfferMessage(Agent a, CN_Task t)
  … abstract int bidDelayTime()
  … abstract CN_Message selectBid(CN_Message[] ms, int l)
  … abstract CN_Task.Award createAward(CN_Task t)
  … abstract CN_Message newAwardMessage(Agent a, CN_Task t)
}
```

Fig. 11. Class `Manager`

Fig. 12 shows the actual sequencing in the `Act()` method of class `Manager`—illustrated by the comments: `Prepare and send offers`, `Wait a while until`

bids have arrived, Collect received bids, Select a bid and prepare and send an award.

```
… void Act() {
  CN_Protocol cnd = (CN_Protocol) protocol;
  CN_Message m;
// … … … … … Prepare and send offers …
  int taskNo = allTasks++;
  for (int i = 0; i<cnd.contractorAgent.length; i++) {
    CN_Task t = createCN_Task(taskNo);
    if (t.addOffer(createOffer(t))) {
      m = newOfferMessage(cnd.contractorAgent[i], t);
      initiateProtocol(m, cnd.contractorRole[i]);
    };
  };
// … … … … … Wait a while until bids have arrived …
  rolePause(bidDelayTime());
// … … … … … Collect received bids …
  CN_Message[] ms = new CN_Message[…];
  int i = 0;
  while ((m = (CN_Message) handleMessage())!=null) {
    if (m.typeMessage(CN_Kind.BID)) {
      ms[i++] = m;
    };
  };
// … … … … … Select a bid, prepare and send an award …
  if ((m = selectBid(ms, i))!=null) {
    CN_Task t = m.cnTask;
    t.addAward(createAward(t));
    CN_Message mm = newAwardMessage(m.fromAgent, t);
    replyProtocol(mm, m);
  };
}
```

Fig.12. Method `Act()` of `Manager`

Fig. 13 shows abstract class `Contractor` as an extension of `Participant`. Method `Act()` of class `Contractor` uses the abstract methods in italics (implemented in `Test_System`).

```
… abstract class Contractor extends Participant {

  … void Act() {…}

  … abstract CN_Task.Bid createBid(CN_Task t)
  … abstract CN_Message newBidMessage(Agent a, CN_Task t)
  … abstract int awardDelayTime()
  … abstract void handleAward(CN_Message m)
}
```

Fig. 13. Class `Contractor`

Fig. 14 shows the actual sequencing in the `Act()` method of class `Contractor`—illustrated by the comments: `Wait to receive an offer`, `Possibly prepare and send a bid`, `Wait to receive an award`, `Possibly receive and handle the award`.

```
… void Act() {
// … … … … … Wait to receive an offer …
  CN_Message m = (CN_Message) roleAwait();
  if (m.typeMessage(CN_Kind.OFFER)) {
// … … … … … Possibly prepare and send a bid …
    CN_Task t = m.cnTask;
    if (t.addBid(createBid(t))) {
      CN_Message mm = newBidMessage(m.fromAgent, t);
      replyProtocol(mm, m);
// … … … … … Wait to receive an award …
      rolePause(awardDelayTime());
// … … … … … Possibly receive and handle the award …
      if ((m = (CN_Message) handleMessage())!=null) {
        if (m.typeMessage(CN_Kind.AWARD)) {
          handleAward(m);
        };
      };
    } else return;
  };
}
```

Fig. 14. Method `Act()` of `Contractor`

Fig. 15 shows `CN_Message` as an extension of `Agent_Message` where `CN_Task` represents the actual task to be undertaken (with respect to `Offer`, `Bid` and `Award`) and `CN_Kind` enumerates the actual message types in Contract Net.

```
… class CN_Message extends Agent_Message {
  … CN_Message(… , CN_Task cnt, CN_Type cnk, …) {
    …
  }
  …
  … CN_Kind cnk;
  … CN_Task cnt;
}

… class CN_Task {…}

enum CN_Kind {OFFER, BID, AWARD}
```

Fig. 15. Classes `CN_Message`, `CN_Task` and `CN_Kind`

6 Test System

Fig. 16 shows `Test_System`, as an extension of `Utility_System` where class `Test_Agent` extends `Utility_Agent`. The abstract methods `newInitiatorRole` and `newparticipantRole` are implemented to return objects of the actual `Manager` and `Contractor` classes specialized from `Manager` and `Contractor` of `Utility_Agent`. Classes `Manager` and `Contractor` implement the abstract methods from Fig. 11 and 13, respectively.

```
… class Test_System extends Utility_System {

  … class Test_Agent extends Utility_Agent  {
    … Initiator newInitiatorRole(Protocol protocol) {
      return (new Manager((CN_Protocol) protocol));
    }
    … Participant newParticipantRole() {
      return (new Contractor());
    }
    …
    … class Manager extends Utility_Agent.Manager {…}
    … class Contractor extends Utility_Agent.Contractor {…}
  }
}
```

Fig. 16. `Test_System` with `Test_Agent`

The protocol using the `OFFER`, `BID` and `AWARD` messages is simple and therefore the structures of the roles illustrated in Fig. 12 and 14 are simple too. But these roles would remain simple if they involved additional interaction, i.e. such as re-announcing subtasks, continued negotiations about details, etc. A `Test_Agent` involved in several simultaneous contract negotiations would not complicate the description but only require additional instantiations of the existing protocol.

Fig. 17 is a snapshot of the dynamic flow of messages between agents. This feature is a part of the application framework, i.e. it is general although it is parameterized with the actual extension of the framework—in this case Contract Net. For each agent, i.e. for `Test_Agent 2` there is a column of messages sent `Messages Out: 6` and received `Messages In: 6` showing total number of messages and a list of actual messages. The actual messages are colored to indicate the status of a message, i.e. *sent*, *received*, *forwarded*, *handled* and *to be removed*. It can be seen from Fig. 17 that $\texttt{TEST\_Agent}_2$ sends `offer 5` that is received by $\texttt{TEST\_Agent}_1$; $\texttt{TEST\_Agent}_1$ replies with `bid 5` that is received by $\texttt{TEST\_Agent}_2$; $\texttt{TEST\_Agent}_2$ replies with `award 5` that is received by $\texttt{TEST\_Agent}_1$. This protocol is similar to the M_1, M_2, M_3 protocol illustrated in Fig. 2.

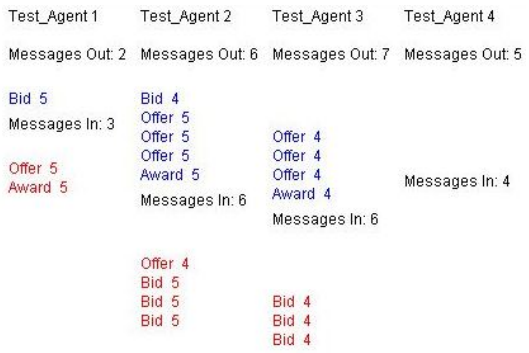


Fig. 17. Flow of messages between agents

7 Background, Related Work, Evaluation

Background. The FLIP project [7] investigates a transportation system including moving boxes from a conveyor belt onto pallets and transporting these pallets in the high bay area of the LEGO® factory with AGVs, no human intervention and only centralized control. A toy prototype includes agents in the form of LEGOBots based on a LEGO® MindstormsTM RCX brick extended with a PDA and wireless LAN. The DECIDE project [8] includes a number of real applications: Control of a baggage handling system in a larger airport in Asia; Intelligent control of handling material with recipes in productions processes; Coordination and planning of large vehicle transports at a shipyard; Design and implementation of a very flexible packing machine. These applications illustrates that the complexity of the communication structure between agents needs to be supported by structurally simple and expressive abstractions.

A course about agent oriented programming includes the construction of a multi-agent system based on the application framework. The task is to design and implement the management of the evolution of a collection of animal parks. A solution is to use reactive roles to react to incoming messages concerning actual changes—and proactive roles to support buying and selling animals by negotiating with other agents. The experience includes that the complexity of the communication structure of simple toy-like multi-agent systems is overwhelming, because the basic communication sequence is simple but the management of several simultaneously ongoing communication sequences is complicated.

Related Work. The application framework is for *implementing* protocols and agent roles, i.e. for *describing* abstractions and *using* these in concrete applications. The purpose of [9] is *modelling* of agent interaction protocols in AUML as a set of UML idioms and extensions. In [10] the purpose is to *specify*, *validate* and *evaluate* interaction protocols expressed as recursive colored petri nets. The purpose of [11] is to *experiment* with the enhancement of object orientation with agent-like interaction including protocol and role introduced in the `powerJava` extension of Java to allow session-aware interactions. In [12] the purpose is to load interaction protocols dynamically through role, action and message ontologies, process description with decision-making rules and a three-layer agent architecture. Dialogue games are the basis for agent interaction protocols for convincing through arguments—in [13] by formal definition of the PARMA protocol—in [14] by a categorization of types of dialogue games with examples of protocols.

The use of object-oriented languages for creating frameworks with concepts from multi-agents is well known, as well as the notion of protocol, agent role, reactive and proactive agents. But the actual form of protocol, reactive and proactive roles of agents and their inclusion in the application framework is original. A protocol is between one initiator and several participants, i.e. the initiator sends a message to the participants that may send a message back to the initiator, i.e. the initiator communicate with each of the participants but the participants do not communicate together. Protocols may be organized with part-protocols to support that a participant (as part of an ongoing protocol) may be initiator of a part-protocol.

Reactive and proactive roles are related to *behaviours* in JADE [15] and *plans* in JACK [16]: In JADE the agent life-cycle is described by behaviors by extending the `Behaviour` class. An agent can execute several behaviors in parallel. However, behavior scheduling is not preemptive, but cooperative—and everything occurs within a single Java thread. In JACK an agent will look for the appropriate plans to handle goals and events. The plan (an abstraction above object-oriented constructs) inherits from a `Plan` class that implements the plan´s base methods and the underlying functionality. Neither behaviors nor plans support the notion of reactive and proactive role explicitly but may be utilized to expose similar behavior. In JADE `createReply()`creates a new message properly setting the receivers and various fields used to control the conversation. In JACK `reply(received, sendBack)` sends a message back to an agent from which a previous message has been received without triggering a new plan.

Evaluation. Each of the *n* agents in the Contract Net example may send an offer to the *n-1* other agents that may reply back etc.:

- Without the protocol abstraction we assume each agent has one (typically reactive) role, i.e. *n* roles in total. However such a role has to manage up to *n* ongoing communications (one of which is between up to *n* agents) each with their own state of the communication. Without the protocol abstraction each role takes care of *n* communications.
- With the protocol abstraction each agent has a manager role and sends an offer to *n-1* other agents each with a contractor role, i.e. in total *n* roles. When *n* agents send an offer this becomes n^2 roles in total. However each role is simple as illustrated in Fig. 12 and 14 because each role is involved in exactly one protocol, i.e. the state of the communication is captured by the role. With the protocol abstraction each role takes care of 1 communication.

In summary the agent model and framework are simple and understandable but still expressive. By the abstractions protocol and agent role we substitute the usual complexity of describing the handling (including state and progress) of several simultaneously ongoing communications by simple, statically structured protocols and roles.

By identifying protocol and agent roles in the Contract Net we classify the interaction and the contributions of the agents by means of `CN_protocol`, `Manager` and `Contractor`. However, abstraction includes not only classification but also specialization and composition: We may see `CN_protocol`, `Manager` and `Contractor` as a general description of the Contract Net, so that specialized versions of Contract Net can be described by specializations of each of these abstractions, e.g. `CN_protocol_X`, `Manager_X` and `Contractor_X`. Similarly, another more extensive protocol can be composed by using the Contract Net as a part protocol by using `CN_protocol`, `Manager` and `Contractor` in the description of this protocol.

Classes `Protocol`, `Initiator` and `Participant` together form abstractions over an interaction structure. `Initiator` and `Participant` are local to an `Agent` in order to have access to the local state of the `Agent`. Alternative solutions may be inspired from [17] where *Association* is a central abstraction over interaction sequences and integrate activities and roles of concurrent autonomous entities.

7 Summary

Typically the communication structure between agents becomes complicated because powerful abstractions are not available for modelling and programming. The application framework with protocols based on agent roles of agents offer abstract description of simple and expressive multi-agent communication structures.

Acknowledgments. We thank Palle Nowack and Daniel May for inspiration and support.

References

[1] M. Wooldridge. An Introduction to Multiagent Systems. Wiley, 2/e, 2009.
[2] G. Booch. Private communication, 2007.
[3] M. E. Fayad, R. E. Johnson, D. C. Schmidt. Building Application Frameworks: Object-Oriented Foundations of Framework Design. Wiley, 1990.
[4] D. Lea. Concurrent Programming in Java: Design Principles and Patterns. Addison Wesley 2007.
[5] O.-J. Dahl, B. Myhrhaug, K. Nygaard. SIMULA 67 Common Base Language (Editions 1968, 1970, 1972, 1984), Norwegian Computing Center, Oslo, 1968.
[6] W. Pree. Meta Patterns: A Means for Capturing the Essentials of Reusable Object-Oriented Design. Proceedings of the 8th European Conference on Object-Oriented Programming (Springer-Verlag), 150–162, 1994.
[7] L. K. Jensen, B. B. Kristensen, Y. Demazeau. FLIP: Prototyping Multi-Robot Systems. Journal of Robotics and Autonomous Systems. Vol. 53, (3, 4), 2005.
[8] K. Hallenborg. Intelligent Control of Material Handling Systems. In: Environmentally Conscious Materials Handling, M. Kutz (edt.), John Wiley & Sons, 2009.
[9] J. Odell, H. Van Dyke Parunak, B. Bauer. Representing Agent Interaction Protocols in UML. First international workshop, AOSE 2000 on Agent-oriented software engineering, 121 - 140, Springer, 2001.
[10] H. Mazouzi, A. El Fallah Seghrouchni, S. Haddad. Open Protocol Design for Complex Interactions in Multi-agent Systems. Autonomous Agents and Multi-Agent Systems, 2002: 517-526.
[11] M. Baldoni, G. Boella, L. van der Torre. Importing Agent-like Interaction in Object Orientation. Proceedings of the 7th WOA Workshop, From Objects to Agents, 2006.
[12] M. Wang, Z. Shi, W. Jiao. Dynamic Interaction Protocol Load in Multi-agent System Collaboration. Multi-Agent Systems for Society. Lecture Notes in Computer Science, Volume 4078, 2009, pp 103-113.
[13] K. Atkinson, T. Bench-Capon, P. McBurney. A Dialogue Game Protocol for Multi-Agent Argument over Proposals for Action. Autonomous Agents and Multi-Agent Systems, 11 (2), 153-171, 2005.
[14] P. McBurney, S. Parsons. Dialogue Games in Multi-Agent Systems. Informal Logic, 22 (3) (2002), pp. 257–274.
[15] F. Bellifemine, G. Caire, D. Greenwood. Developing Multi-Agent Systems with JADE. Wiley, 2008.
[16] http://www.agent-software.com. JACK Intelligent Agents—Agent Manual. JACK Intelligent Agents—Agent Practicals.
[17] B. B. Kristensen. Rendezvous-Based Collaboration between Autonomous Entities: Centric versus Associative. Concurrency and Computation: Practice and Experience, vol. 25, no. 3, pp. 289-308, Wiley Press, 2013.

A Namespace Approach for Modularity in BDI Programming Languages

Gustavo Ortiz-Hernández[1,2], Jomi F. Hübner[3], Rafael H. Bordini[4], Alejandro Guerra-Hernández[2], Guillermo J. Hoyos-Rivera[2], and Nicandro Cruz-Ramírez[2]

[1] Centro de Investigaciones en Inteligencia Artificial - UV
Xalapa, México
[2] Institute Henri Fayol - MINES
Saint-Étienne, France
[3] Federal University of Santa Catarina
Florianópolis, SC, Brazil
[4] FACIN-PUCRS
Porto Alegre, RS, Brazil

Abstract. In this paper we propose a model for designing Belief-Desire-Intention (BDI) agents under the principles of modularity. We aim to encapsulate agent functionalities expressed as BDI abstractions into independent, reusable and easier to maintain units of code, which agents can dynamically load. The general idea of our approach is to exploit the notion of *namespace* to organize components such as beliefs, plans and goals. This approach allowed us to address the name-collision problem, providing interface and information hiding features for modules. Although the proposal is suitable for agent-oriented programming languages in general, we present concrete examples in Jason.

Keywords: Agent-Oriented Programming, Modularity, Namespace

1 Introduction

In the last decades, several programming paradigms have arisen, often presented as an evolution of their predecessors, and with the main purpose of abstracting more complex and larger systems in a more natural and simpler way. Particularly, the Agent-Oriented-Programming (AOP) paradigm has been promoted as a suitable option to deal with the challenges arising when developing modern systems. This paradigm offers high-level abstractions which facilitate the design of large-scale and complex software systems, and also allows software engineers to employ a suite of well-known strategies for dealing with complexity, i.e., decomposition, abstraction and hierarchy.

These strategies are usually applied at the Multi-Agent-System (MAS) level [18,7,13]. However, even a single agent is intrinsically a complex system, hence its design and development should consider the above mentioned strategies. Regarding this, the principle of *modularity* applied to individual agent development can significantly improve and facilitate the construction of agents.

In this paper, we present an approach for programming agents following the principle of modularity, i.e., to develop agent programs into separate, independent, reusable and easier to maintain units of code. In order to support modularity, we identify three major issues to be addressed: i) a mechanism to avoid name-collision, ii) fulfilling the information hiding principle, and iii) providing module interfaces.[5]

Our contribution is to address these issues by simply introducing the notion of *namespace* in the AOP paradigm. In the context of BDI languages, which is the focus of this paper, the novelty of our approach is that it offers a syntactic level solution, independent of the operational semantics of some language in particular, which simplifies its implementation.

The rest of this paper is organized as follows: related and previous work are presented in Section 2; our proposal is described in Section 3; we explain details of implementation in Section 4 and offer an example in Section 5; an evaluation is presented in Section 6; finally, we discuss and conclude in Sections 7 and 8 respectively.

2 Related Work

There exist much work supporting and implementing the idea of modularity in BDI languages. An approach presented by Busetta et al. [5] consists in encapsulating beliefs, plans and goals that functionally belong together in a common scope called capability. The programmer can specify a set of scoping rules to say which elements are accessible to other capabilities. An implementation is developed for JACK [16]. Further, L. Braubach et al. [2] extend the capability concept to fullfill the information hiding principle by ensuring that all elements in a capability are part of exactly one capability and remain hidden from outside, guaranteeing that the information hiding principle is not violated. An implementation for JADEX [3] is provided. Both approaches propose an explicit import/export mechanism for defining the interface.

The modules proposed by Dastani et al. [11] are conceived as separate mental states. This modules are instantiated, updated, executed and tested using a set of predefined operations. Executing a module means that the agent starts reasoning in a particular mental state until a predefined condition holds. This approach is extended by Cap et al. [6], by introducing the notion of sharing scopes to mainly enhance the interface. Shared scopes allow modules posting events, so that these are visible to other modules sharing the same scope. These ideas are conceived in the context of 2APL [9] and an implementation is described in [10].

Also following the notion of capability, Madden and Logan [20] propose a modularity approach based on XML's strategy of namespaces [4], such that each module is considered as a separate and unique namespace identified by an URI. They propose to handle a local belief-base, local goal-base and local events-queue for each module, and then to specify, by means of an export/import statement,

[5] In general, these are well-known issues when building programming languages as discussed in [5,2,20].

which beliefs, goals and events are visible to other modules. In this system, there is only one instance of each module, i.e., references to the exported part of the module are shared between all other modules that import it. These ideas are supported by the Jason+ language, implemented by Logan and Kiss [8].

Another work tackling the name-collision issue is presented by G. Ortiz et al. [14]. They use annotations to label beliefs, plans and events with a source according to the module to which they belong. In this approach, modules are composed by a set of beliefs, plans and a list of exported and imported elements. Both imported and exported elements are added to a unique common scope. An implementation of this approach is developed as a library that extends Jason.

In Hindriks [15], a notion of module inspired by what they call policy-based intentions is proposed for GOAL. A module is designated with a mental state condition, and when that condition is satisfied, the module becomes the agent focus of execution, temporarily dismissing any other goal. They focus on isolating goals/events to avoid the pursuit of contradictory goals.

In Riemsdijk et al. [23], modules are associated with a specific goal and they are executed only to achieve the goal for which they are intended to be used. In this approach, every goal is dispatched to a specific module. Then all plans in the module are executed one-by-one in the pursuit of such goal until it has been achieved, or every plan has been tried. This proposal is presented in the context of 3APL [12].

A comparative overview of these approaches is given in Table 1. All solutions tackle the name-collision problem, providing a mechanism to scope the visibility of goal/events to a particular set of elements, e.g., plans. They also offer different approaches for providing the interface of modules. However, not all of them fulfill the information hiding principle.

It is also worth mentioning that all those approaches propose some particular operational semantics tied to the AOP language in which they have been conceived and implemented. The proposal that we present in this paper provides a mechanism to address those issues independently of the operational semantics.

3 Modules and Namespaces

A *module* is as a set of beliefs, goals and plans, as a usual agent program, and every agent has one initial module (its initial program) into which other modules can possibly be loaded. We refer to the beliefs, plans and goals within a module as the *module components* (cf. Figure 1).

Modularity is supported through the simple concept of *namespace*, defined as an abstract container created to hold a logical grouping of components. All components can be prefixed with an explicit namespace reference. We write `zoo::color(seal,blue)` to indicate that the belief `color(seal,blue)` is associated with the namespace identified by `zoo`. Furthermore, note that the belief `zoo::color(seal,blue)` is not the same belief as `office::color(seal,blue)` since they are in different namespaces.

approach	IL	IS	IH	NC	interface's mechanism
Busetta et al. [5]	JACK	✗	✓	✓	explicit import/export
Braubach et al. [2]	JADEX	✗	✓	✓	explicit import/export
Dastani et al. [11]	2APL	✗	✗	✓	set of predefined operations
Cap et al. [6]	2APL	✗	✗	✓	sharing scopes
Madden et al. [20]	Jason+	✗	✓	✓	explicit import/export
Hindriks [15]	GOAL	✗	✗	✓	mental-state condition
Riemsdijk et al. [23]	3APL	✗	✗	✓	goal dispatching
Ortiz et al. [14]	Jason	✗	✗	✓	unique-common scope
Our Proposal	Jason	✓	✓	✓	global namespaces

Table 1: The columns represent existing features in the surveyed approaches, in respect to the issues mentioned in Section 1. The abbreviations stand for: (IL) implementing language; (IS) the approach is independent of the language's operational semantics; (IH) fulfills the information hiding principle; (NC) provides a mechanism to deal with the name-collision issue. The last column refers to the general notion used to provide an interface.

Namespaces are either *global* or *local*. A global namespace can be used by any module; more precisely, the components associated with a global namespace can be consulted and changed by any module. A local namespace can be used only by the module that has defined the namespace.

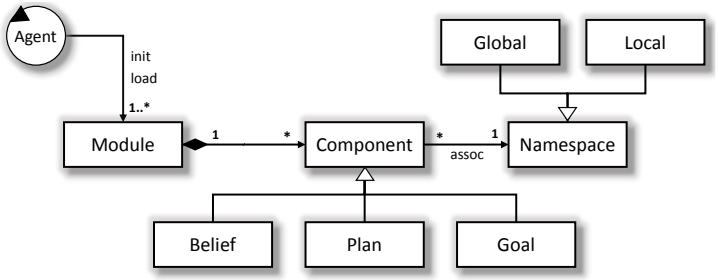


Figure 1: Proposed model for modularity.

We introduce the notion of *abstract namespace* of a module to denote a namespace whose name is undefined at design-time, and will be defined at run-time when the module is loaded. To indicate that a component is in a module's abstract namespace, the prefix is simply omitted, e.g., a belief written as `taste(candy,good)` is in an abstract namespace and its actual namespace will be defined when the module is loaded.

The module loading process involves associating every component in the abstract namespace of the module with a concrete namespace, and then simply incorporating the module components into the agent that loaded the module.

Therefore, a concrete namespace must be specified at loading time to replace the module's abstract namespace.

When a module (the loader) loads another module (the loaded), they interact in two directions: the loader *imports* the loaded module components associated with global namespaces and the loader *extends* the functionality of the module by placing components in those namespaces. Figure 2 illustrates the interaction when a module A loads some module B.

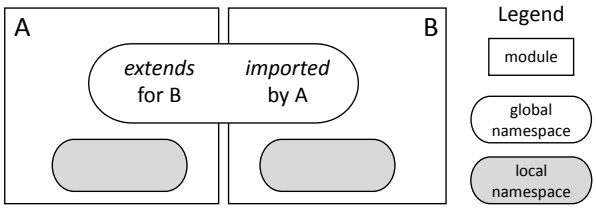


Figure 2: The interaction between modules.

A module is formally defined as a tuple:

$$mod = \langle bs, ps, gs \rangle$$

where $bs = \{b_1, \ldots, b_n\}$ is a set of beliefs, $ps = \{p_1, \ldots, p_n\}$ is a set of plans, and $gs = \{g_1, \ldots, g_n\}$ a set of goals. As shown in Figure 1, each of these components is associated with a namespace. We use subscripts to denote the elements of a module, e.g., mod_{bs} stands for the beliefs included in module mod.

3.1 Syntax

As in many programming languages, we use identifiers to refer to the module components, i.e., its beliefs, plans and goals. Since the syntactic identifiers depend on the programming language and our proposal is intended to be language independent, we propose to extend the syntax of identifiers allowing a namespace prefix:

$$\langle id \rangle ::= [\ \langle nid \rangle ::]\ \langle natid \rangle$$

where *nid* is a namespace identifier and *natid* is used to denote the native identifiers of some AOP language. For example, a belief formula like `count(0)`, whose identifier is `count`, can be written `ns2::count(0)` to associate the belief with namespace `ns2`.

We use a syntactical name-mangling technique[6] to associate every component in the abstract namespace of a particular module to a concrete namespace and to bring support for local namespaces. Restriction access to local namespaces is implemented by replacing every local namespace identifier in the components of

[6] A technique used in programming languages, which consists in attaching additional information to an identifier, typically used to solve name conflicts.

Algorithm 1: The $mangling(src, nid)$ function associates each component in the abstract namespace of a module program src with a concrete namespace nid and renames local namespaces with an internally generated identifier.

```
1 begin
      Input: src : a module program
      Input: nid : a concrete namespace
2     mod = parse(src)
3     foreach id ∈ ids(mod) do
4         if ns(id) is an abstract namespace then
5             replace id by nid::id in mod
6         if ns(id) is local then
7             replace id by #nid::id in mod
8     return mod
```

a particular module by an internally created identifier. This is generated in such a way that it is not a valid identifier according to the grammar of the native language. For instance, if `ns2` is the identifier of a local namespace, the mangling function renames `ns2::color(box,blue)` to `#ns2::color(box,blue)`, where `#ns2` is an invalid identifier and thus no developer can write a program that accesses this belief. We use `#`nid to denote a mapping from nid to an internally generated identifier, unique in the module program where it is being used. The mangling function is described in algorithm 1. To avoid cluttering the notation, we define an auxiliary function $ids(mod) = \{id_1, \ldots id_n\}$ that gets all identifiers id that are in the components bs, ps, and gs of module mod and function $ns(id)$ gives the namespace of identifier id.

3.2 Loading Modules

We represent an agent state as a tuple $ag = \langle B, P, G, \ldots \rangle$, where $B = \{b_1, \ldots, b_n\}$ stands for the agent's belief base, $P = \{p_1, \ldots, p_n\}$ a plan library and $G = \{g_1, \ldots, g_n\}$ the goals of the agent[7]. All identifiers used in the beliefs, plans and goals are prefixed with a proper namespace. The dots symbol $(\ldots)$ is used in the agent tuple to denote the existence of other components proper of the agent's mental state (such as intentions, mail box, etc.) that are not relevant for the purpose of presenting our proposal.

When an agent loads a module, it incorporates the module components, i.e., beliefs, plans and goals, into its own belief base, plan library and goals,

[7] Sometimes when referring to intentional agents, a distinction between desires and intentions is highlighted to focus on the commitment of the agent towards some goal. In the agent state we do not take commitment into consideration; a goal $g \in G$ can be either a desire or an intention. However, a goal $g \in gs$ in some module is considered as an initial goal.

respectively. A namespace must be specified at loading time to replace the module's abstract namespace with a concrete namespace. A transition rule (**Load**) presents the dynamics of loading a module in a particular namespace. The condition (upper part) stands for the action `load(`src`,`nid`)` that takes a module program src and a namespace nid as parameters. This rule executes the mangling function on the module and incorporates the module components into the agent's current state, already associated with a proper namespace identifier.

$$(\textbf{Load})\frac{\texttt{load(}src\texttt{,}nid\texttt{)}}{\langle B, P, G, \ldots\rangle \rightarrow \langle B', P', G', \ldots\rangle}$$

$$\begin{aligned} \text{where:} \quad mod &= mangling(src, nid) \\ B' &= B \cup mod_{bs} \\ P' &= P \cup mod_{ps} \\ G' &= G \cup mod_{gs} \end{aligned}$$

The agent's *initial module* is loaded in what we call the *default namespace*. This is a predefined global namespace whose identifier is `default`. The initial module program determines the initial belief base, plan library and goals of the agent. We use src_0 to denote the initial module program. The next transition rule (**Init**) describes the agent's initialization.

$$(\textbf{Init})\frac{src_0}{\langle B, P, G, \ldots\rangle \rightarrow \langle B', P', G', \ldots\rangle}$$

$$\begin{aligned} \text{where:} \quad mod &= mangling(src_0, \texttt{default}) \\ B' &= mod_{bs} \\ P' &= mod_{ps} \\ G' &= mod_{gs} \end{aligned}$$

4 Implementation

We present the implementation of our proposal in Jason [1], a Java-based interpreter for an extended version of AgentSpeak(L) [21]. An agent program in Jason is defined as a set of initial beliefs bs, a set of initial goals gs and a set of plans ps, where each $b \in bs$ is an atomic grounded formula (initial beliefs may also be represented as Prolog style rules). Every plan $p \in ps$ has the form $te : ctx \leftarrow body$, where te stands for a triggering event defining the event that the plan is useful for handling. A plan is considered relevant for execution when an event which matches its trigger element occurs, and applicable when the condition ctx holds. The element $body$ is a finite sequence of actions, goals and belief updates. Actions in Jason can be external or internal. An external action changes the environment, unlike an internal action which is executed internally to the agent. Jason allows the developer to extend the parsing of source code by implementing user-customized directives.

The basic syntactical construct of a Jason program is a literal, which as in logic programming has the form $p(t_1, \ldots, t_n)$, where p is the predicate (that can

be strongly negated with the $\sim$ operator), $n \geq 0$ (literals with 0 arguments are called atoms), and each t_i denotes a term that can be either a number, list, string, variable, or a structure that has the same format of a positive literal. We say then that each predicate p in a Jason program is a Jason identifier. For instance, a plan such as:

```
+!go(home) : forecast(sunny) ← walk_to(0,0).
```

contains the following identifiers: `go`, `home`, `forecast`, `sunny` and `walk_to`.

We have extended the syntax of Jason identifiers to allow a namespace prefix[8]. Since Jason identifiers are used for beliefs and goals, by prefixing them with a namespace these components are *scoped* within a particular namespace[9]. So, a plan written as:

```
+!ns1::go(home) : ns2::forecast(sunny) ← +b.
```

will consider only an achievement-goal addition event `+!go(home)` in namespace `ns1`, and a belief `forecast(sunny)` in namespace `ns2`; beliefs and goals in other namespaces are thus not relevant for this plan. Terms within a literal are not changed when a module is loaded. However, terms can be explicitly prefixed with a namespace. A term prefixed with `::` is in the abstract namespace (e.g. in `forecast(::sunny)` the term sunny is associated with the abstract namespace).

Jason keywords (e.g., `source`, `atomic`, `self`, `tell`, ...), strings and numbers are handled as constants and are not associated with namespaces.

The Jason internal action `.include` and parsing directive `include` were extended with a second parameter to implement the dynamics of loading a module as presented in Section 3.2. The first argument is the file with the module's source code and the second argument the global namespace used to replace the abstract namespace. A parsing directive `namespace/2` is provided to define the type of the namespace (local or global) and as a syntactic sugar to facilitate the namespace association of components, so that the identifiers enclosed by this directive will be associated with the specified namespace.

The beliefs related to perception are placed in the default namespace, and thus also the corresponding events (external events generated from perception). This solution keeps backward compatibility with previous source code, since the initial module is loaded in the default namespace and the previous version of Jason does not have modules other than the initial one.

5 Example

This section illustrates our proposal for modules in more detail showing an implementation of the Contract Net Protocol (CNP) [22]. The modules `initiator`

[8] For the unification algorithm used by Jason, we can simply consider the namespace prefix as being part of the predicate symbol of the literal.

[9] Plans are also scoped within a namespace given that their triggering events are based on beliefs or goals.

and `participant` (Codes 5 and 6) encapsulate the functionality to start and participate in a CNP, respectively. The multi-agent system is composed of the initiator agents `bob` and `alice`, whose initial module code is presented in codes 1 and 2 respectively; and the participant `company A` and `company B` (Codes 3 and 4). In this implementation, every CNP instance takes place in a different namespace to isolate the beliefs and events of each negotiation.

Agent `bob` statically loads the module initiator twice (lines 1-2) using the directive `include/2`. This agent starts two CNP's for tasks `build(park)` and `build(bridge)` (initial goals in lines 4-5) in namespaces `hall` and `comm`. Each goal is handled by the module instance loaded in the same namespace where the goal is posted.

```
1 {include("initiator.asl",hall)}
2 {include("initiator.asl",comm)}
3
4 !hall::startCNP(build(park)).
5 !comm::startCNP(build(bridge)).
6
7
8
9
10
11
12
```

Code 1: bob.asl

```
1  !start([fix(tv),fix(pc),fix(oven)]).
2
3  +!start([]).
4  +!start([fix(T)|R])
5   <- .include("initiator.asl",T);
6      .add_plan(
7      {+T::winner(W)<-
8       .print("Winner to fix",T,"is",W)
9      });
10     // sub-goal with new focus
11     !!T::startCNP(fix(T));
12     !start(R).
```

Code 2: alice.asl

Agent `alice` starts multiple CNP's. It uses the internal action `.include/2` for dynamically loading the module `initiator`. It starts one CNP for each task in a given list of tasks (line 5). Agent alice *extends* the functionality provided by the module initiator to print in the console the winner as soon as it is known. Namely, it adds one plan to the same namespace where the module is loaded (lines 6-9).

Agent `company A` participates in all CNPs. It loads the module `participant` in every namespace where it listen that a CNP has started (note that the namespace in line 2 of code 3 is a variable). The agent customizes the module by adding beliefs in the same namespace where the module is loaded (lines 3-4). The module uses these beliefs to decide what tasks can be accepted and how much to bid (cf. lines 6-7 of Code 6).

Agent `company B` plays participant only for CNPs started by agent `bob`, and taking place in namespaces `hall` or `comm`. When a CNP starts under these conditions, it loads the module `participant` in the corresponding namespace. The beliefs in lines 8-9 and the plan added in lines 14-19 extend the functionality of the module by setting the strategy for bidding and accepting tasks. This company only accepts tasks for building and its bid depends on the namespace in which the CNP is being carried on. Directive `namespace/2` in line 1 defines the local namespace `supp`. This namespace encapsulates the beliefs used to estimate the final price of tasks (lines 2-5), so that they are inaccessible to other modules.

The `initiator` module provides functionality to start a CNP. It starts with a forward declaration of the local namespace `priv` in line 1. The namespace

```
1  +N::cnpStarted[source(A)]
2   <- .include("participant.asl", N);
3      +N::price(_,(3*math.random)+10);
4      +N::acceptable(fix(_));
5      !N::joinCNP[source(A)].
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

Code 3: company A.asl

```
1  {begin namespace(supp,local)}
2     price(bridge,300).
3     price(park,150).
4     gain(hall,1.5).
5     gain(comm,0.8).
6  {end}
7
8  hall::acceptable(build(_)).
9  comm::acceptable(build(_)).
10
11 +N::cnpStarted[source(bob)]
12  : .member(N,[hall,comm])
13  <- .include("participant.asl",N);
14     .add_plan({
15        +?N::price(build(T),P)
16           : supp::gain(N,G)
17           <- ?supp::price(T,M);
18              P=M*(1+G)
19     });
20     !N::joinCNP[source(bob)].
```

Code 4: company B.asl

of `startCNP` (line 11) is abstract and a concrete namespace is given when the module is loaded (cf. lines 1-2 and 5 of Codes 1 and 2, respectively). Because the namespace given to `startCNP` is global (as defined by the loader), this module is *exporting* the plan `@p1`. The identifiers without an explicit namespace between lines 30 and 55 will be placed in the local namespace `priv`. This is used to encapsulate the module's internal functionality, so that the plans to carry out contracts and announcements are only accessible from within this module (as illustrated in the line 23). Similarly, the beliefs added to memorize the current state of the CNP and the rule in lines 4-8 are private and will not interfere or clash with any other belief of the agent. However, a loader module can retrieve the current state of the CNP by means of plans `@p2` and `@p3`. Figure 3 illustrates the relation (imports and extends) between the modules `alice` and `initiator` using the same notation of Figure 2.

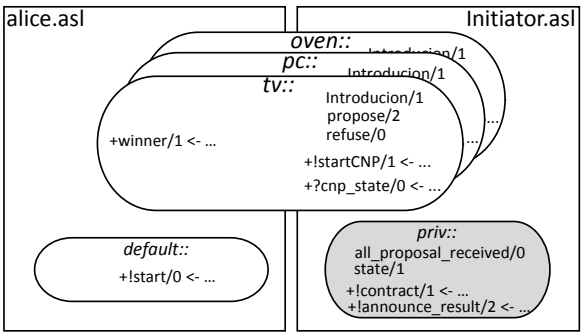


Figure 3: The namespaces of agent alice during its execution.

```
1  {namespace(priv,local)} //Forward definition
2
3  // operator :: forces a term to be considered in the abstract namespace
4  priv::all_proposals_received
5   :- .count(::introduction(participant)[source(_)],NP) &
6      .count(::propose(_)[source(_)], NO) &
7      .count(::refuse[source(_)], NR) &
8       NP = NO + NR. // participants = proposals + refusals
9
10 // starts a CNP
11 @p1 +!startCNP(Task)
12   <- .broadcast(tell, ::cnpStarted); // tell everyone a CNP has started
13      // this_ns is a reference to the namespace where this module is loaded
14      // in this example is the namespace where the CNP is being performed
15      .print("Waiting participants for task ",Task," in ",this_ns,"...");
16      .wait(3000); // wait participants introduction
17      +priv::state(propose); // remember the state of the CNP
18      .findall(A, ::introduction(participant)[source(A)], LP);
19      .print("Sending CFP for ",Task," to ",LP);
20      .send(LP,tell, ::cfp(Task)); //send call for proposals to participants
21      // wait until all proposals are received for a maximum 15secs
22      .wait( priv::all_proposals_received, 15000,_);
23      !priv::contract(this_ns).
24
25 // to let the agent to query the current state of the CNP
26 @p2 +?cnp_state(S) <- ?priv::state(S).
27 @p3 +?cnp_state(none).
28
29 {begin namespace(priv)}
30   //.intend(g) is true if the agent is currently intending !g
31   +!contract(Ns) : state(propose) & not .intend(::contract(_))
32     <- -+state(contract); // updates the state of CNP
33        .findall(offer(Price,A), Ns::propose(Price)[source(A)], L);
34        .print("Offers in CNP taking place in ",Ns," are ",L);
35         L \== [];  // constraint the plan execution to at least one offer
36        .min(L,offer(WOf,WAg)); // sort offers, the first is the best
37        +Ns::winner(WAg);
38        !announce_result(Ns,L);
39        -+state(finished).
40
41   // nothing todo, the current phase is not propose
42   +!contract(_).
43   -!contract(Ns)
44     <- .print("CNP taking place in ",Ns," has failed! None proposals");
45        -+state(finished).
46
47   +!announce_result(_,[]).
48   // announce to the winner
49   +!announce_result(Ns,[offer(_,Ag)|T]) : Ns::winner(Ag)
50     <- .send(Ag,tell, Ns::accept_proposal); // notify the winner
51        !announce_result(Ns,T).
52   // announce to others
53   +!announce_result(Ns,[offer(_,Ag)|T])
54     <- .send(Ag,tell, Ns::reject_proposal);
55        !announce_result(Ns,T).
56 {end}
```

Code 5: initiator.asl

The `participant` module has a plan to join a CNP by sending an introduction message to the agent playing initiator in the corresponding namespace. When a call for proposals is received, an offer is sent back only if the task is supposed to be accepted, otherwise the agent replies with a refuse message (lines 6-13). The accepted tasks and the amount to bid are not provided in the module (lines 6, 7 and 13). They are meant to be defined by a loader module that can

extend every instance of this module to specify both tasks to be accepted and the strategy for bidding (e.g. as in modules `company A` and `company B`).

```
// participating in CNP
+!joinCNP[source(A)]
  <- .send(A,tell, ::introduction(participant)).

// Answer to Call For Proposal
+cfp(Task)[source(A)] : acceptable(Task)
  <- ?price(Task,Price);
     .send(A,tell, ::propose(Price));
     +participating(Task).

+cfp(Task)[source(A)]  :  not acceptable(Task)
  <- .send(A,tell, ::refuse);
     .println("Refusing proposal for task ", Task, " from Agent ", A).

// Answer to My Proposal
+accept_proposal : participating(Task)
  <- .print("My proposal in ",this_ns," for task ", Task," won!").
     // do the task and report to initiator
+reject_proposal : participating(Task)
  <- .print("I lost CNP in ",this_ns," for task ",Task,".").
```

Code 6: participant.asl

6 Evaluation

We developed a non-modular version of the CNP to compare with the version presented in Section 5. Then, we performed five extensions to both versions. The first consists in modifying the vocabulary used by agents for communication. The second modifies the protocol so that every agent specifies the limit of CNP's in which it is able to participate simultaneously. In the third, initiator agents set a deadline for the call for proposals. The fourth adds one more agent playing initiator and four participants with their own acceptable tasks and strategy to bid. Finally, in the fifth only acceptable proposals are announced.

The comparison among the versions is shown in Table 2. The abbreviations stand for: (num) extension number; (ags) number of agents; (I) number of agents playing initiator; (P) number of agents playing participant; (eds) the number of files edited; (m) modular version, i.e., developed using our approach; (n) non-modular version; (adds) blocks of code added; (dels) blocks of code deleted; (chgs) changes in a line of code. The size of the implementation was calculated after compressing the source files with a zip utility. The initial size is given in bytes, then a percentage representing the increment is given. The extensions are progressive and each is compared against the previous.

For instance, to accomplish extension 2 of the modular version (starting from extension 1), we added six blocks of code and changed two lines across a total of four files, which increased the size of the system programs in 8.2% (i.e., 190 and 195 more bytes than initial implementation and extension 1, respectively) when compared with its previous extension. To extend the corresponding non-modular version, three files were edited to add twelve blocks of code and perform

num	extension	ags		eds		size		updates					
								adds		dels		chgs	
		I	P	*m*	*n*	*m*	*n*	*m*	*n*	*m*	*n*	*m*	*n*
	initial implementation	2	3	-	-	2359	2864	-	-	-	-	-	-
1	update communication vocabulary	2	3	2	5	-0.5%	0.8%	0	0	0	0	15	37
2	participants set a limit of CNP's	2	3	4	3	8.2%	7.0%	6	12	0	0	2	6
3	initiators set a deadline	2	3	3	2	2.1%	1.5%	3	4	0	0	1	2
4	add more participants	3	7	5	5	50.6%	85.9%	48	126	0	0	0	0
5	participants are not notified if lose	3	7	2	10	-1.3%	-6.6%	0	0	2	10	0	0
	total	-	-	16	25	59.1%	88.6%	57	142	2	10	18	45

Table 2: Comparison of the CNP across a series of extensions.

six changes in different lines, increasing the program size in 7% (i.e., 224 and 199 more bytes than initial implementation and extension 1, respectively). The number of agents remained the same in both versions. Total row summarizes the updates and the increase along all extensions of the system. If the same file had to be edited during two different editions it is counted twice.

The results show that the modular version required a total of 77 updates (57 additions, 2 deletions and 18 changes) against the non-modular for which 197 updates were necessary. In this particular case study we are reducing the maintainability effort by 60% (120 updates less). We can conclude that a project developed using our approach is easier to maintain.

This results can be analyzed in terms of the Don't repeat yourself (DRY) principle[10]. Our proposal enforces this principle since it represents a mechanism to avoid the repetition of code in several parts of the system. In contrast to the non-modular version, where every component implementing the functionality of the protocol is repeated in the program of each agent, the higher the number of participant agents (interested in different tasks and having distinct bidding strategies) the greater the count of repetition occurrences. For instance, if some change is performed in the protocol, even as simple as the way in which participants introduce themselves, the change have to be propagated to the source code of every agent participating in the CNP's.

We made some initial effort in comparing with a version using the usual `include` directive in previous releases of Jason, but for the chosen metrics and example the difference to a version with no includes appeared negligible so we kept the values for the latter. In future work we will consider other metrics and examples where the difference to a version with the old include directive might be more significant. In any case, it should be emphasized that clearly the old

[10] A principle of software development with the purpose of reducing the repetition of information [17], so that a modification of any single element of a system does not require a change in other logically unrelated elements.

directive does not solve the problem of name collision nor supports information hiding. For instance, if an agent `tom` already uses `price/2` (e.g., to record the prices for supplies), when it includes the source file implementing the CNP (using an include without support for namespaces), since the belief `price/2` is also used by the CNP implementation to determine the bids, a name-collision arises and the resulting behavior is unexpected[11]. For solving this, it is necessary either to change the name of the belief used by `tom` to record the prices of supplies, or that one used in the CNP implementation. Note that the latter alternative implies updating every agent using the source file implementing the CNP.

The following section overviews how this proposal for modules addresses the issues mentioned in Section 1; and highlight some of its properties, as well as the major differences of our approach over related work mentioned in Section 2.

7 Discussion

The notion of namespaces adapted to the context of BDI-AOP languages is suitable to address the main issues related to modularity. For instance, the name-clashing problem is tackled by associating each component to a unique namespace, enabling the programmer to write qualified names for disambiguating references to components.

The interface is provided through the concept of global namespace, which supports both importing components and extending the functionality of modules. The notion of abstract namespace allows dynamic association of module components to namespaces, thus the same module can be loaded several times in different namespaces and also multiple modules can be loaded into the same namespace to compose a more complex solution. The local namespaces permit programmers to encapsulate components which among other things facilitates independent development of modules. Moreover, loading modules at runtime can be seen as dynamic updating, i.e., the acquisition of new capabilities without stopping its execution.

The main difference of our approach resides in the strategy adopted to achieve modularity. On the one hand, the strategy in this paper consists in logically organizing component names in the agent's mental state, by attaching additional information to their identifiers. On the other hand, the approaches mentioned in Section 2, in general, are based in mechanisms for dealing with multiple mental states inside the agent, in which modules are active components of the operational semantics of the language, i.e., new transition rules are needed for handling multiple belief bases, plan libraries and/or event queues in the same reasoning cycle. The latter strategy leads to solutions that are more difficult to implement, in contrast to ours, which brings a syntactic level solution, so that it can be implemented in any BDI language by simply extending their parsers.

[11] This is also reported by N. Madden and B. Logan [20] from the experience of using the usual `include` directive available in previous releases of Jason for the development of a large-scale multi-agent system [19].

8 Conclusion

In this paper we have presented a solution for programming BDI Agents under the principles of modularity, and we explored the assumption that the notion of namespace is enough to address the main issues related to modularity, such as avoiding name-collisions, following the information hiding principle and providing an interface. We have exemplified the properties and feasibility of the approach using the Jason language.

It is future work to provide an unload mechanism that removes components from modules that are no longer used by the agent. We also aim to implement the approach in other languages to further evaluate the generality of the approach.

References

1. Rafael H. Bordini, Jomi F. Hübner, and Michael J. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons, Ltd, 2007.
2. Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. Extending the capability concept for flexible BDI agent modularization. In *Proceedings of the Third international conference on Programming Multi-Agent Systems*, ProMAS'05, pages 139–155, Berlin, Heidelberg, 2006. Springer-Verlag.
3. Lars Braubach, Er Pokahr, and Winfried Lamersdorf. Jadex: A BDI agent system combining middleware and reasoning. In *Ch. of Software Agent-Based Applications, Platforms and Development Kits*, pages 143–168. Birkhaeuser, 2005.
4. Tim Bray, Dave Hollander, Andrew Layman, and Richard Tobin. Namespaces in XML 1.0. W3C recommendation, W3C, August 2006. Published online on August 16th, 2006 at `http://www.w3.org/TR/2006/REC-xml-names-20060816`.
5. Paolo Busetta, Nicholas Howden, Ralph Rönnquist, and Andrew Hodgson. Structuring BDI agents in functional clusters. In Nicholas R. Jennings and Yves Lespérance, editors, *Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL), 6th International Workshop, ATAL 99, Orlando, Florida, USA, July 15-17, 1999, Proceedings*, volume 1757 of *Lecture Notes in Computer Science*, pages 277–289. Springer, 1999.
6. Michal Cap, Mehdi Dastani, and Maaike Harbers. Belief/goal sharing BDI modules. In *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 3*, AAMAS '11, pages 1201–1202, Richland, SC, 2011. International Foundation for Autonomous Agents and Multiagent Systems.
7. Pedro Cuesta, Alma Gomez, and JuanCarlos Gonzalez. Agent oriented software engineering. In Antonio Moreno and Juan Pavon, editors, *Issues in Multi-Agent Systems*, Whitestein Series in Software Agent Technologies and Autonomic Computing, pages 1–31. Birkhäuser Basel, 2008.
8. Bryan Logan Daniel Nicholas Kiss. Jason+ – extension of the jason agent programming language. Technical report, School of Computer Science and Information Technology, University of Nottingham, 2010.
9. Mehdi Dastani. 2APL: A practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, June 2008.
10. Mehdi Dastani, Christian P. Mol, and Bas R. Steunebrink. Modularity in agent programming languages an illustration in extended 2APL. In *In the proceedings of The 11th Pacific Rim International Conference on Multi-Agents (PRIMA), Springer, 2009*, pages 139–152. LNCS, 2009.

11. Mehdi Dastani and Bas Steunebrink. Modularity in BDI-based multi-agent programming languages. In *Proc. of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology - Volume 02*, WI-IAT '09, pages 581–584, Washington, USA, 2009. IEEE Computer Society.
12. Mehdi Dastani, Birna van Riemsdijk, Frank Dignum, and John-Jules Ch. Meyer. A programming language for cognitive agents: Goal directed 3APL. In Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors, *Programming Multi-Agent Systems*, volume 3067 of *LNCS*, pages 111–130. Springer, 2004. 1st International Workshop, PROMAS 2003, Melbourne, Australia, July 15, 2003.
13. Marie-Pierre Gleizes Federico Bergenti and Franco Zambonelli. Methodologies and software engineering for agent systems. In Federico Bergenti, Marie-Pierre Gleizes, and Franco Zambonelli, editors, *The Agent-Oriented Software Engineering Handbook*, volume 11 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 4–10. Springer, 2004.
14. Ortiz-Hernández G., Guerra-Hernández A., and Hoyos-Rivera G. D. J. JasMo - a modularization framework for Jason. IEEE, 12th Mexican International Conference on Artificial Inteligence (MICAI). Mexico City, November 2013.
15. Koen Hindriks. Modules as policy-based intentions: modular agent programming in GOAL. In *Proc. of the 5th international conference on Programming multi-agent systems*, ProMAS'07, pages 156–171, Berlin, Heidelberg, 2008. Springer-Verlag.
16. N. Howden, R. Ronnquist, A. Hodgson, and A. Lucas. JACK intelligent agents - summary of an agent infrastructure. In *Proceedings of the 5th ACM International Conference on Autonomous Agents*, 2001.
17. Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
18. Nicholas R. Jennings. Agent-oriented software engineering. In Ibrahim Imam, Yves Kodratoff, Ayman El-Dessouki, and Moonis Ali, editors, *Multiple Approaches to Intelligent Systems*, volume 1611 of *Lecture Notes in Computer Science*, pages 4–10. Springer Berlin Heidelberg, 1999.
19. N. Madden and B. Logan. Collaborative narrative generation in persistent virtual environments. In *Intelligent Narrative Technologies: Papers from the 2007 AAAI Fall Symposium*, Menlo Park, CA, November 2007. AAAI Press.
20. Neil Madden and Brian Logan. Modularity and compositionality in Jason. In Lars Braubach, Jean-Pierre Briot, and John Thangarajah, editors, *Programming Multi-Agent Systems: 7th International Workshop, ProMAS 2009, Budapest, Hungary, May 10-15, 2009. Revised Selected Papers*, volume LNAI 5919, pages 237–253, Budapest, Hungary, 2010. Springer, Springer.
21. Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : agents breaking away*, MAAMAW '96, pages 42–55, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.
22. R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Trans. Comp.*, 29(12):1104–1113, 1980.
23. M. Birna van Riemsdijk, Mehdi Dastani, John-Jules Ch. Meyer, and Frank S. de Boer. Goal-oriented modularity in agent programming. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, AAMAS '06, pages 1271–1278, New York, NY, USA, 2006. ACM.

ARGO: A CUSTOMIZED JASON ARCHITECTURE FOR PROGRAMMING EMBEDDED ROBOTIC AGENTS

Carlos Eduardo Pantoja[1,4], Márcio Fernando Stabile Junior[2], Nilson Mori Lazarin[1], and Jaime Simão Sichman[3]

[1] Centro Federal de Educação Tecnológica (CEFET/RJ), Brazil
{pantoja, nilson.lazarin}@cefet-rj.br
[2] Instituto de Matemática e Estatística, Universidade de São Paulo, Brazil
mstabile@ime.usp.br
[3] Escola Politécnica, Universidade de São Paulo, Brazil
jaime.sichman@poli.usp.br
[4] Universidade Federal Fluminense, Brazil

Abstract. This paper presents ARGO, a customized Jason architecture for programming embedded robotic agents using the Javino middleware and perception filters. Jason is a well known agent-oriented programming language that relies on the Belief-Desire-Intention model and implements an AgentSpeak interpreter in Java. Javino is a middleware that enables automated design of embedded agents using Jason and it is aimed to be used in the robotics domain. However, when the number of perceptions increases, it may occur a bottleneck in the agent's reasoning cycle since an event is generated for each single perception processed. A possible solution to this problem is to apply perception filters, that reduce the processing cost. Consequently, it is expected that the agent may deliberate within a specific time limit. In order to evaluate ARGO's performance, we present some experiments using a ground vehicle platform in a real-time collision scenario. We show that in certain cases the use of perception filters is able to prevent collisions effectively.

1 Introduction

Agents are autonomous and pro-active entities situated in an environment and are able to reason about what goal to achieve, based on its perceptions about the world [19]. In robotics, an agent is a physical entity composed of hardware, containing sensors and actuators, and software that is responsible for its reasoning. The Belief-Desire-Intention model (BDI) [3] is a cognitive approach for reasoning based on how information from the environment and the goals an agent has can activate predefined plans in order to try to achieve these goals. Jason [2] is an Agent-Oriented Programming Language (AOPL) that implements an AgentSpeak interpreter in Java, adopting the BDI cognitive architecture. However, programming robotic agents using Jason is a difficult task because a

bottleneck can occur in the agent's reasoning cycle when the robot updates its belief base with perceptual information.

Javino [11] is a middleware that enables automated design of embedded agents using Jason. It allows agents to communicate with microcontrollers in hardware devices, e.g. Arduino. Both Javino and Jason can run embedded in a single-board computer such as Raspberry Pi (connected with n devices). However, when using several sensors, the agent's belief base generates events for each perception, which may compromise the robot execution time. In [17], perception filters were used to minimize the cost effects of processing all perceptions in simulation systems using Jason. The results showed that filters are able to improve agent's performance significantly.

Thus, in this paper, we present a customized Jason architecture for programming embedded robotic agents named ARGO[5], which uses a layered robot architecture separating the hardware from the reasoning agency. In ARGO, Javino enables processing data coming from sensors as perceptions in ARGO's agent reasoning cycle. Then, one can restrict the list of perceptions delivered by Javino based on filters designed by the agent's programmer. The main contribution of ARGO is to enable the use of perception filters for programming robotic agents, which reduces the cost of processing perceptions in BDI. Moreover, ARGO allows an agent to decide when to start or to stop perceiving, to fix the interval between each perception and to control the perceptual behavior by using Jason internal actions to filter perceptions at runtime.

In order to evaluate ARGO's performance, we also present some experiments using a ground vehicle platform in a real-time collision scenario constructed. We applied the experimental design methodology described by [10] to test and to statistically verify that in certain cases the use of perception filters reduces BDI processing time, thus preventing collisions effectively.

The rest of the paper is structured as follows. We briefly present in section 2 the Jason framework (subsection 2.1), the Javino middleware's structure and functionality (subsection 2.2) and then explain how we can construct embedded robotic agents with these frameworks (subsection 2.3). In the sequence, perception filters are discussed in section 3. In this moment, we are able to present ARGO architecture and its implementation in section 4. Our experiments are presented in section 5, where we define the case study, the experimental design and show our results. In section 6, we discuss related work. Finally, in section 7 we present our conclusions and further research directions.

2 Programming BDI agents

2.1 Jason

Jason [2] is an interpreter for an extended version of AgentSpeak [14], which is an abstract AOPL based on a restricted first-order language with events and actions. Created to allow the specification of BDI agents, Jason implements the

[5] Download available at http://argo-for-jason.sourceforge.net.

operational semantics of AgentSpeak and provides a platform for the development of multi-agent systems.

A Jason agent operates by means of a reasoning cycle that is analogous to the BDI decision loop [2]. First, the agent receives a list of literals representing the current state of the environment. Then, the belief base is updated based on the perceptions received. Each change in the belief base generates an event that is added to a list to be used in a posterior step. The interpreter checks for messages that might have been delivered to the agent's mailbox. These messages go through a selection process to determine whether they can be accepted by the agent or not. After that, one of the generated events is chosen to be dealt with and when it is selected, all the plans related to that event are selected. From these plans, a new selection is made to separate which of them can be executed given the current state of the environment. If more than one plan can be executed, a function selects which one will be executed. If the agent has many different foci of attention, a function chooses one intention among those for execution. The final step is to execute the first non-executed action from the selected intention.

2.2 Javino

Javino is a library for both hardware and software that implements a protocol for exchanging messages between the low-level hardware (microcontrollers) and the high-level software (programming language) with error detection over serial communication [11]. There are some communicating libraries in the literature, such as RxTx Library and JavaComm, based on serial ports. However, these libraries do not provide error detection and they use byte-to-byte communication. In both cases, the programmer needs to implement a message controller on the hardware layer in order to avoid losses.

The format of a message used in a communication by Javino is composed of 3 fields: preamble, size and message content. The preamble (2 bytes) identifies the beginning of a new message that arrived through a serial port. The size field (1 byte) is calculated before any transmission informing the size of the message. The field message content (up to 255 bytes) carries the message that has to be sent.

Both the preamble and size fields identify errors in case of loss or collision of information during the message transmission. When a message arrives on the serial port, the receiver (either software-side or hardware-side) verifies the preamble. If it is correct, the receiver then counts the size of the message content field and compares it with the value of size field: if they don't match, the message is discarded. In the case of incomplete messages, the receiver also discards the message. Javino provides three different operation modes:

– the **Send** Mode assumes a simplex message transmission by software to hardware. It uses the *sendCommand(port, msg)* method to send a message to the hardware-side. This method returns a boolean value which gives a feedback about the successful transmission to the microcontroller. This feedback is

necessary because the port serial can be locked by other concurrent transmissions. The software-side do not wait for answers from the hardware;
- the **Request** Mode assumes a half-duplex transmission between software to hardware, where the hardware sends an answer message. It uses the *requestData(port, msg)* method, that sends a message to the hardware-side through a serial port and returns a boolean value which checks if there is any answer sent by the hardware-side. The user is supposed to implement an answer message in the hardware-side using the *availableMsg()* method, that verifies if it exists a valid message from software-side, the *getMsg()* method, that gets the message sent by software-side and the *sendMsg(msg)* method, that sends a message to software-side;
- the **Listen** Mode assumes a simplex transmission by hardware to software. It uses the *listenHardware(port)* method to check if there is any message sent by the hardware-side. The Request and Listen modes get messages from hardware using the *getData()* method.

The Javino's protocol aims to be multi-platform and can be implemented using any programming language. The hardware-side library may be used in microcontrollers such as ATMEGA, PIC or Intel families. The software-side library may be coded in Java or in another programming language. In [11], it was developed a Java library for the software-side and an Arduino library for hardware-side. In this case, Javino requires both Python and pySerial installed to manage the serial port of an operational system.

2.3 Embedding robotic agents

Some previous research have tried to integrate robotic reasoning into hardware by using BDI agents. In [7], a framework was presented to provide a way of programming agents using AgentSpeak in Unmanned Aerial Vehicle in a simulator. The authors in [4] proposed an aquatic robot which uses Arduino together with BeagleBoard who could move from point-to-point deviating from obstacles. However, the reasoning was centralized on a computer using a Wi-Fi communication with the robot. All the decisions were sent to BeagleBoard and retransmitted, by serial communication, to Arduino, which held sensors and actuators. Another work published in [1] presented a grounded vehicle, which used Arduino and Jason to control sensors and actuators using a Java library for communication between the hardware and the Jason's environment. However, the agent's reasoning was still running on the computer. The messages to the hardware-side were sent from an Arduino connected to a USB port computer to another Arduino embedded on the robot using radio transmitters.

The work in [16] showed that it was possible to use BDI agents on embedded systems employing single-board computers. However, it was not presented an infrastructure to integrate BDI agents in a robot. Therefore, they simulated the environment on a computer to execute the decisions taken by the BDI agent.

Finally, a robotic agent platform using both Javino and Jason framework was presented in [11], which was an improvement of the platform presented in

[1]. The authors used Raspberry Pi and Arduino together to provide a fully embedded BDI agent reasoning on a robot. In this case, Javino was integrated into the agent's simulated environment and the agent used a Jason external action to request the perceptions and a Jason internal action to control the actuators. In this architecture, the agent is responsible for controlling both sensors and actuators that are connected to the Arduino board and it is embedded in Raspberry Pi. The Arduino boards are connected to the USB ports of Raspberry Pi, thus, the agents use Javino to get perceptions from sensors and act with the actuators plugged in Arduino. The architecture worked in embedded robotic agents. However, according to the authors, when using too many sensors or plans in Jason code the agent's reasoning suffered a delay due to the cost of processing perceptions in Jason. We believe that using filters to overcome this issue could reduce the time employed in perceptions processing in BDI.

3 Perception filters in Jason

In order to identify the critical points for performance in the Jason reasoning cycle, the work in [17] used a profiling tool to analyze a piece of Jason code. By measuring memory and CPU usage, the authors verified that two sections of the code were more time-consuming: the Belief Update Function (BUF) and the method responsible for the *unification* of variables in the plans and rules. These two methods generated a bottleneck, and depending on the specification of the agent, those methods could take up to 99% of reasoning time.

Given that Jason's default implementation assumes that everything that an agent can perceive in the environment will be part of its perception list, they proposed the inclusion of a perception filter between the perceive function and the update of beliefs before starting the reasoning cycle. This filter is responsible for analyzing the perception list received and for removing from the list those literals that are not interesting for the agent. This is done through filters defined by the agent designer which are described in XML format files and define restrictions on the predicate, variables and annotations of the beliefs.

Let us suppose a robotic agent that represents his beliefs about the environment by predicates like $p(d, v)$, where predicate p identifies the sensor, d the side of the robot where the sensor is located and v the value acquired by perception. An example of perception list would be:

```
temperature(right,36)
temperature(back,38)
light(left,143)
distance(front,227)
distance(right,30)
```

An example of filter that is used in the experiments section 5.1 is shown below. This filter would remove all the perceptions originated from the temperature and light sensors and would also remove the perceptions from the distance sensors that are not in the front of the robot.

```
<?xml version="1.0"?>
<PerceptionFilter>
   <filter>
      <predicate>temperature</predicate>
   </filter>
   <filter>
      <predicate>light</predicate>
   </filter>
   <filter>
      <predicate>distance</predicate>
      <parameter operator="NE" id="0"> front </parameter>
   </filter>
</PerceptionFilter>
```

Since the agent's intentions may change, the perceptions that are relevant for the agent may also change. To reflect these changes, a new Jason internal action called *change_filter* was also proposed in [17]. This action receives as a parameter the name of an XML file with the specific rules for the perceptions, and sets it as the current filter so that in the next reasoning cycle, the agent receives perceptions according to its new interests.

4 ARGO

In this section, we present ARGO, a customized architecture that employs both Javino middleware and perception filters for programming embedded robotic agents using the Jason framework. Javino provides a bridge between the intelligent agent and the robot's sensors and actuators while perception filters act by blocking specific perceptions coming from Javino. Furthermore, we also present a layered architecture for constructing cognitive robots.

4.1 Overview of a Robot's Architecture using Javino

A robotic agent is an embedded system where software and hardware components are integrated to provide sensing and operating abilities in real-time environments. For this, it is necessary to employ an architecture capable of facilitating the robot construction and programming. Hence, we propose an architecture for programming robotic agents where it is possible to design the robot platform independently from the reasoning agency, and then to integrate them using a protocol for serial communication.

The robot platform must be composed of sensors and actuators coupled to microcontrollers, where all the desired actions that the robot can perform in the environment and the percepts it can capture from sensors are programmed. In this case, our architecture translates raw data into a format for high-level programming language in the firmware, resulting in a performance gain for the agent's reasoning. Javino's protocol is responsible for sending these percepts using the serial port of the microcontroller. In this architecture, it is possible

to use any kind of microcontrollers whereas it employs a library compliant with Javino's protocol. Afterwards, a MAS programming language is employed to allow the cognitive control of the robot platform. The chosen program language should be able to host the existing versions of Javino's protocol or to implement a new one. An overview of the architecture is shown in Figure 1.

The architecture is composed of three layers: hardware, firmware and reasoning. The hardware layer is responsible for mounting the robot platform, sensors, actuators and connecting them with respective microcontrollers employed. A single-board computer is used to connect all microcontrollers using USB and will be responsible for hosting the MAS. The firmware layer provides all actions that a robot can execute including procedures for both sensors and actuators and they are programmed directly in the microcontroller. Basically, these procedures send prepared raw data as percepts for the reasoning layer and receive agent's messages to perform some action, both using serial communication and the hardware-side of Javino's protocol.

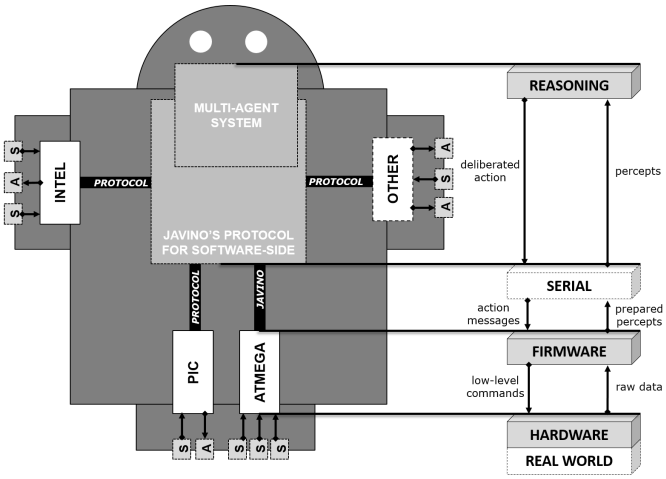


Fig. 1. Overview of a robot's architecture using Javino.

The reasoning layer represents the MAS's programming using a high-level language. The middleware in software-side transmits received percepts from serial port to the agent and sends action messages to the firmware layer. Depending on the AOPL chosen, it is possible to integrate received percepts directly into the agent's reasoning cycle or to use some structure to control the perception flow. As the architecture allows many microcontrollers in a robot platform, a strategy for capturing those percepts should be implemented. For example, it is possible to read all available serial ports one by one and after that to update the agent's percepts or to allow the agent decide which serial port it desires to use at a particular moment. Note that an agent cannot access more than one serial

port at a time and more than one agent cannot access the same serial port at the same time.

In most of the commercial platforms, programmers do not have access to implementation details or they have to use an interface as a middleware for controlling the robot; on the other hand, these platforms also present a suite of functions to help in robot motion and planning. Our approach aims to be an architecture for open robot design to be used in cases where the programmer needs freedom to build his own prototype, using open platforms such as Arduino. The architecture is not bound neither to the MAS programming language, which can be interchanged, nor to the hardware adopted. However, it is necessary to adjust the raw data translation to percepts in the firmware layer, if the AOPL is changed.

4.2 ARGO

In the reasoning layer of our proposed robot architecture, it is necessary to adopt an AOPL which will be responsible for the cognitive reasoning of the robot platform. For this, we propose a customized Jason's architecture named ARGO employing perceptions filters and Javino integrated into Jason's agent's reasoning cycle.

ARGO aims to be a practical architecture for programming automated embedded agents using BDI agents in the robotics domain. An ARGO agent is able to directly control the actuators at runtime and it receives perceptions from the sensors automatically within a pre-defined time interval. It is also able to change its filters at runtime based on its needs (the same can occur when accessing its devices). The BDI in Jason implies a high cost of processing the perceptions since for each one of the received literals an event is generated. In complex codes, plans may be added in running time, and a quite large intention stack is generated. In these cases, if the robotic agent has to achieve a goal within a time limit, it may not succeed. Our idea is to apply perception filters in these cases, so as to enable the agent to deliberate in time, in order to act in such critical applications. An overview of ARGO can be seen in Figure 2.

An agent can assume to be an ARGO agent by defining the Argo architecture in the MAS design; otherwise, the standard agent architecture of Jason is automatically defined. An ARGO agent is supposed to connect to one or more devices at runtime by choosing which serial port it wants to access (until the limit of 127 serial ports); however it can only use one port at a time, for both sensing and acting. Besides that, different ARGO agents must not use the same serial port at the same time, because when exists a competition for communicating at the same port, there is data loss. An ARGO agent is able to communicate with others common Jason Agents, but only ARGO agents can control devices and receive percepts from the environment. In this case, ARGO agents send the received percepts and can delegate for Jason agents the reasoning about these perceptions if it is desirable. Once the agent has received percepts, it can filter them based on its actual configuration.

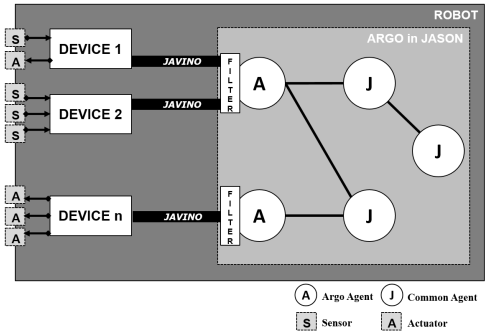


Fig. 2. ARGO overview.

4.3 Internal actions

An ARGO agent can also decide when to perceive the real world at runtime. It means that the agent can start and stop perceiving from sensors when it is desirable. Similarly, it can also directly control the actuators by using internal actions. The Argo architecture also provides ways of filtering perceptions by using a specific internal action, where it is defined the name of an XML file that holds the perceptions conditions. So, we propose five internal actions for programming agents in Jason along with Argo architecture:

1. **limit(x):** defines the sensing interval, where x is a value in milliseconds;
2. **port(y):** defines which serial port should be used by the agent, where y is a literal representing the port identification, e.g. COM8;
3. **percepts(open—block):** decides whether or not to perceive the real world;
4. **act(w):** sends to the hardware an action, represented by literal w, to be executed by a microcontroller;
5. **change_filter(filterName):** defines the filter to constrain perceptions in runtime, where filterName is the name of the XML file containing the filter constraints.

4.4 Customizing Jason for ARGO

In Jason's reasoning cycle, as mentioned in section 2.1, the agent gets its percepts from the simulated environment provided by Jason. We extended the reasoning cycle of Jason, shown in Figure 3, to providing a customized architecture for ARGO agents. First, Javino middleware is now responsible for getting percepts coming from low-level layers and sends them to the perceive step. Before being incorporated in the belief base, percepts can be filtered based on the agent's active filter. Then, filtered perceptions are processed and the reasoning cycle flows up to the act step, where the agent can perform basic Jason's actions or an action to control the actuators of the robot, which once more involves Javino middleware.

In order to create ARGO architecture, it was necessary to customize Jason framework, in particular by extending the *AgArch* class. This class is responsible for the Jason's native architecture and provides a list of perceptions sent by the Jason's environment in Java and the communication with other agents [2]. In the extended architecture, Javino middleware was inserted as a communication bridge to the hardware sensors and actuators. Besides that, the serial port identification had to be added to the native *AgArch* class in order to define to which serial port the Javino has to communicate.

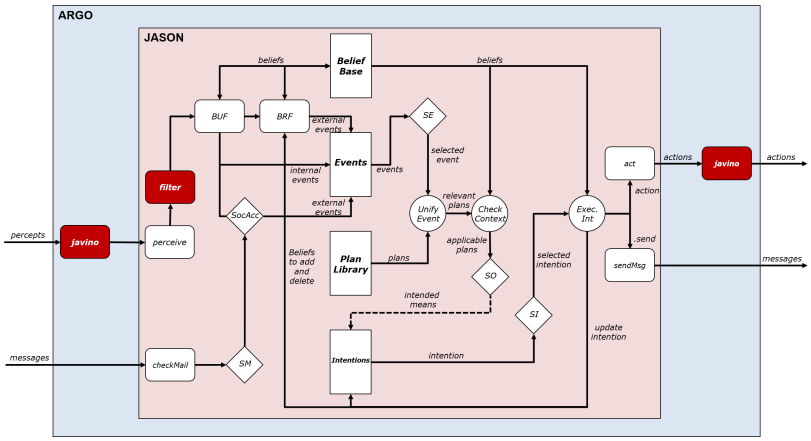


Fig. 3. ARGO reasoning cycle.

In the *TransitionSystem* class, two new attributes *blocked* and *limit* were created, as well as a new function *realWorldPerceptions*. The blocked attribute is responsible for blocking or unblocking the perceptions and the limit attribute specifies a time interval for perceiving the real world (data from sensors). The *realWorldPerceptions* verifies in each cycle *(i)* if the percepts are blocked; or *(ii)* if the time limit for the next perception has been reached. If the percepts are not blocked and the time limit was reached, Javino requests the percepts from sensors and sends them to the perceive method in Agent class.

Before the agent processes the percepts coming from Javino, they can be filtered using the method *filter* also implemented in the Agent class. In this case, all agents have the ability to filter percepts, because this method was implemented in the native Agent class. The modifications executed do not change Jason's original functionality, except for the simulated environment which is not used since Javino gets the percepts from the real world. We opted for creating a customized architecture instead of an infrastructure because the later one obliges all agents to be ARGO agents.

5 Experiments

5.1 Case study

In order to evaluate the overall architecture and to assure the impact of the perception filter, we assembled a robot composed of four distance sensors, four light sensors, four temperature sensors, an Arduino board and an Arduino 4wd chassis. A sensor of each type was placed in each of the four sides of the robot (front, back, right and left). The robot was placed on a flat surface two meters away from a wall. When started, the robot would perceive the environment and move forward at a constant speed[6] until the distance to the wall was less than a specified value. As soon as it perceived that the distance was smaller, the robot should stop. The robot can be seen in Figure 4.

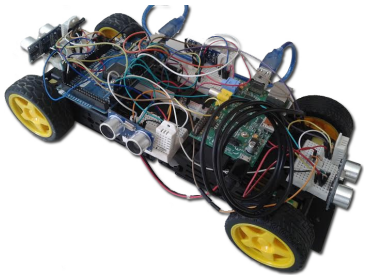


Fig. 4. The robot used in the experiments.

5.2 Experiment design

The experiment presented was designed based on the experimental design guidelines presented in [10]. According to the author, the goal of a proper experimental design is to obtain the maximum information with the minimum number of experiments. The procedure separates the effects of various factors that might affect the performance and allows to determine if a factor has a significant effect or if the observed difference is simply due to random variations caused by measurement errors and/or parameters that were not controlled. It is important to define the meaning of four terms:

1. *Response Variable* is the outcome of an experiment. In the experiments executed, the response variables are the processing time taken by the agent to stop after perceiving the wall and the distance it stopped from the wall;
2. *Factors* are the variables that affect the action response variable. Factors can be Primary or Secondary. Primary factors are those whose effects need to

[6] The speed is about 10 cm/s and it is not used in the experiments since it is constant.

be quantified while secondary factors are those that impact the performance but whose impact we are not interested in quantifying. The primary factors chosen for this experiments were the distance the agent should stop from the wall, the time interval for receiving the perceptions and the filter used;
3. *Levels* are the values that a factor can assume. The factors and levels used are presented in Table 1;
4. *Replication* is the repetition of all or some experiments. If all experiments in a study are repeated three times, the study is said to have three replications.

Factor	**Levels**		
Distance	40 cm	80 cm	120 cm
Perception interval	20 ms	35 ms	50 ms
Filter	No filter	Front Side	Front Distance

Table 1. Factors and levels used for the experiment

The three filter levels represent the filter configurations that were used. "No filter" represents that the ARGO architecture did not make use of the perception filters, "Front Side" represents that the filter removed all the perceptions, except the ones from the sensors present on the front side of the robot. "Front Distance" represents that the filter removed all the perceptions, except the ones from the distance sensor present on the front side of the robot. Three executions were conducted for every combination of levels in Table 1.

5.3 Results

The first response variable analyzed was the distance the agent stopped from the wall. Figure 5 shows the results of all possible value combinations of the different factors presented in Table 1. Bars that do not appear in the Figure mean that the agent collided with the wall.

One should notice initially that in all cases, the agent that didn't filter its perceptions collided with the wall (there is no any blue bar in the Figure). In some cases, for instance, the experiment for distance 120 cm, the agent with front side filter arrived eventually to stop before the wall (with perception intervals 20 and 35), but always closer to the wall when compared to the agent that used front distance filtering. This agent outperformed the others in quite all the experiments, and it was able to successfully stop before hitting the wall in all the experiments when the distance limit was 80 cm or 120 cm. However, in some experiments (for example, distance 40 cm and perception interval 50), neither agent could avoid the collision.

The second response variable analyzed was the time taken by the agent to stop after perceiving the wall. For this experiment, we calculated the variation assigned to each factor, as detailed in [10]. This statistical analysis is useful to

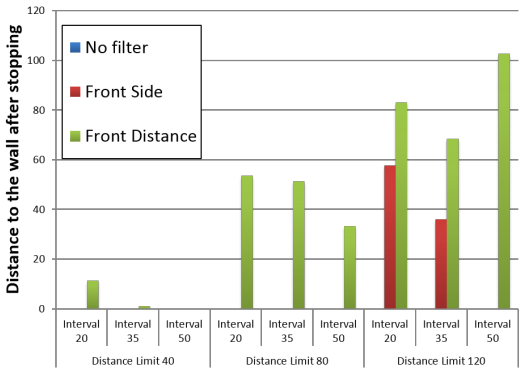


Fig. 5. Distance to the wall after stopping.

Factor	**Variation attributed**
Distance Limit (L)	1,415%
Perception Interval (I)	0,165%
Filter (F)	88,965%
Interaction between L and I	0,525%
Interaction between L and F	3,715%
Interaction between I and F	0,265%
Interaction between L and I and F	1,725%
Error	3,28 5%

Table 2. Variation assigned to each factor in the analysis of the response time.

check which one of the factors is being responsible for the differences in the response variable. The calculated values are presented in Table 2.

The results confirm the importance of the filter in reducing the processing time since almost all variation was attributed to it. This result suggests that ARGO architecture, by integrating Javino and the perception filters, can be used for developing embedded robotic agents in a way that the agent can benefit from the BDI architecture with a smaller influence of one of its major drawbacks that would be the high processing time.

6 Related Work

Robot architectures deal with platforms, sensors, actuators, programming language and reasoning mechanisms. One challenge is how to integrate these components in a way that a robot can deliberate to perform a task without failing to accomplish its goal. In [18] it is proposed a cognitive control architecture integrating knowledge representation of sensory and cognitive reasoning of a robotic agent using GOAL. The architecture consists of four decoupled layers: robot platform, robot behavioral control, environment interface and cognitive control. The

robot platform employed was the humanoid NAO and it used URBI as middleware for interfacing with the robot's hardware via TCP/IP protocol. The robot behavioral control layer is responsible for processing sensory data and monitoring and executing behaviors. Besides, this layer communicates (using TCP/IP) with the reasoning and the robot platform layer, transmitting sensory data and actions execution respectively. The interface layer uses a translation mechanism between the sensory information acquired from the behavioral layer and the percepts sent to the cognitive layer. This layer is necessary because symbolic and sub-symbolic information can use different languages. The mechanism is based on a standard template using XML files mapping, which indicates how to map data but also when to do it. The cognitive control layer uses GOAL [8], which is a logic-based programming language for cognitive agents.

Similarly, ARGO's architecture also divides the robot programming into layers, separating sensory data from the agent's reasoning. We exploit the advantages of Jason extending it for programming robotic agents. ARGO provides three layers to be programmed: hardware, firmware and agent reasoning. Our proposed architecture provides a support for exchanging the hardware and the firmware without concerning with reasoning layer; furthermore, it is possible to change the agent programming language without changing either the hardware or the firmware. This is possible because Javino is responsible for exchanging serial messages between these layers, and it does not link them to each other. We do not provide a translation mechanism in high-level layers because of the processing cost, which can affect the robot efficiency. However, the translation from raw data into percepts is done in the firmware layer. Since ARGO aims to be used in open platforms, the programmer must code the firmware layer. For commercial platforms such as NAO and Lego Mindstorms, a percepts mapping process must be provided.

Some other works also use Jason for this end, such as [13] and [12]. In [13], CArtAgO [15] is used as the functional layer for providing artifacts that represent sensors and actuators of a robot, and Jason is used as the reasoning layer. Despite using artifacts, which is an interesting abstraction for the devices employed, the authors use a simulator named Webots and do not embed the MAS. [12] provides a Jason extension for ROS named Rason.

Javino's protocol provides a mechanism for avoiding noisy data in communication between the firmware and the reasoning layer. However, we do not treat noisy data coming directly from sensors, when they provide well-formed but wrong values. In [6], it is presented a programming language for cognitive robots and software agents using 3APL [9] language, which implements a deliberation cycle for selecting and executing practical reasoning rule statements and goal statements. It also provides an architecture consisting of beliefs, goals, actions and practical reasoning rules as a mental state. The beliefs represent the robots percepts of an environment. The authors focused only on the programming constructs, they do not provide information about how a robot platform should interact with the high-level language.

In [5], it is presented a Teleo-Reactive (TR) extension for programming robots, supported by a double tower architecture which provides a percepts handler that atomically updates the BeliefStore (a repository of beliefs). After that, it reconsiders all rules affected by this change. The authors assert that actions and percepts can be dispatched through ROS interface to the robot platform. TR extension uses low-level procedures written in procedural programming languages for sensorial data and actuators actions. It was used Qu-Prolog, simulators and Lego Mindstorms robots.

ARGO has the same intention of facilitating the programming of robotic agents providing a mechanism for automatically updating the agents belief base. The TR extension provides an inhibition process of some behaviors in response to percepts while ARGO provides a runtime process for filtering percepts that are not needed at a specific moment. Filtering perceptions in ARGO prevents unnecessary event triggering in the deliberative cycle of Jason, therefore, the agent deliberation should be more efficient.

7 Conclusions and further work

This paper presented ARGO, an architecture for programming embedded robotic agents using Jason framework, which uses the Javino middleware for exchanging serial messages through a serial port and perceptions filters. Javino allows a mechanism for perceiving from sensors and process these perceptions directly in the agents' reasoning cycle. When many perceptions are processed, Jason generates a stack of events that delays the decision in real-time situations. Perceptions filters proved to be a solution to overcome this situation.

In the experiments, we show that applying the perception filter together with Javino reduces the time of processing perceptions significantly in Jason. In a real-time collision scenario, where the agent had to reason and stop before colliding with an obstacle placed at 120cm, 80cm and 40cm, the experiment showed there only by using the perception filters the agent was able to stop before colliding. The ARGO architecture aims to provide programming structures that allow coding robotic agents using Jason. It means that an agent can decide when to act and to perceive at runtime. Furthermore, it is able to change perceptions filters based on its needs, and to decide what device it will be connected to at a certain time during its execution.

For future work, we intend to extend ARGO architecture for programming multi-robot systems through a communication protocol between robotic agents. Moreover, it is necessary to test ARGO in different domains and apply robotics technics such as SLAM. We will also intend to provide other hardware-side libraries, for instance for PIC and Intel families.

Acknowledgments

Márcio F. Stabile Jr. is financed by CNPq. Carlos Pantoja is financed by CAPES. Jaime Simão Sichman is partially financed by CNPq, proc.303950/2013-7.

References

1. Barros, R.S., Heringer, V.H., Lazarin, N.M., Pantoja, C.E., Moraes, L.M.: An agent-oriented ground vehicles automation using Jason framework. In: 6^{th} International Conference on Agents and Artificial Intelligence. pp. 261–266 (2014)
2. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons Ltd (2007)
3. Bratman, M.E.: Intention, Plans and Practical Reasoning. Cambridge Press (1987)
4. Calce, A., Forooshani, P.M., Speers, A., Watters, K., Young, T., Jenkin, M.R.: Autonomous aquatic agents. In: ICAART (1). pp. 372–375 (2013)
5. Clark, K., Robinson, P.: Robotic agent programming in TeleoR. In: Robotics and Automation, 2015 IEEE International Conference on. pp. 5040–5047 (2015)
6. Dastani, M., de Boer, F., Dignum, F., Van Der Hoek, W., Kroese, M., Meyer, J.J., et al.: Programming the deliberation cycle of cognitive robots. In: Proc. of the 3rd International Cognitive Robotics Workshop (2002)
7. Hama, M.T.: Uma plataforma orientada a agentes para o desenvolvimento de software em veículos aéreos não-tripulados. Master's thesis, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil (2012)
8. Hindriks, K.V.: Programming rational agents in GOAL. In: Seghrouchni, A., Dix, J., Dastani, M., Bordini, H.R. (eds.) Multi-Agent Programming: Languages, Tools and Applications, pp. 119–157. Springer US, Boston, MA (2009)
9. Hindriks, K.V., De Boer, F.S., Van der Hoek, W., Meyer, J.J.C.: Agent programming in 3APL. Autonomous Agents and Multi-Agent Systems 2(4), 357–401 (1999)
10. Jain, R.: Art of Computer Systems Performance Analysis: Techniques For Experimental Design Measurements Simulation and Modeling. Wiley (2015)
11. Lazarin, N.M., Pantoja, C.E.: A robotic-agent platform for embedding software agents using raspberry pi and arduino boards. In: 9^{th} Software Agents, Environments and Applications School (2015)
12. Morais, M., Meneguzzi, F., Bordini, R., Amory, A.: Distributed fault diagnosis for multiple mobile robots using an agent programming language. In: Advanced Robotics (ICAR), 2015 International Conference on. pp. 395–400 (2015)
13. Mordenti, A., Ricci, A., Santi, D.I.A.: Programming robots with an agent-oriented bdi-based control architecture: Explorations using the jaca and webots platforms. Bologna, Italy, Tech. Rep (2012)
14. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: de Velde, W.V., Perram, J.W. (eds.) Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world (MAAMAW'96). Lecture Notes in Artificial Intelligence, vol. 1038, pp. 42–55. Springer-Verlag, USA (1996)
15. Ricci, A., Piunti, M., Viroli, M., Omicini, A.: Environment programming in CArtAgO. In: Seghrouchni, A., Dix, J., Dastani, M., Bordini, H.R. (eds.) Multi-Agent Programming: Languages, Tools and Applications, pp. 259–288. Springer US, Boston, MA (2009)
16. Santos, F.R., Hübner, J.F., Becker, L.B.: Concepção e análise de um modelo de agente BDI voltado para o planejamento de rota em um VANT. In: 9^{th} Software Agents, Environments and Applications School (2015)
17. Stabile Jr., M.F., Sichman, J.S.: Evaluating perception filters in BDI Jason agents. In: 4^{th} Brazilian Conference on Intelligent Systems (BRACIS) (2015)
18. Wei, C., Hindriks, K.V.: An agent-based cognitive robot architecture. In: Dastani, M., Hübner, J.F., Logan, B. (eds.) Programming Multi-Agent Systems: 10th International Workshop, ProMAS, Valencia, Spain, pp. 54–71. Springer, Berlin (2013)
19. Wooldridge, M.J.: Reasoning about rational agents. MIT press (2000)

A Multi-Agent Solution for the Deployment of Distributed Applications in Ambient Systems

Ferdinand Piette[1,2], Costin Caval[1], Cédric Dinont[2], Amal El Fallah Seghrouchni[1], and Patrick Tailliert[1]

[1] Sorbonne Universités, UPMC Univ Paris 06, LIP6 – Paris – France
[2] Institut Supérieur de l'Électronique et du Numérique – Lille – France

Abstract. In this paper, we present a multi-agent solution for the configuration, deployment and monitoring of distributed applications in the context of Ambient Intelligence (AmI). We describe the use of goal-driven agents and show how the agent organisation allows for the privacy of the infrastructure resources to be enhanced. We illustrate the functioning of the resulting multi-agent system (MAS) through a video doorkeeper in a smart environment scenario, in which a distributed application is deployed several times in different contexts. The agents collaborate to find the right hardware entities in the environment, taking into account the context.

Keywords: Applicative paper, Multi-agent system, Ambient Intelligence, Goal-driven agents, Agent design, Privacy management, Deployment

1 Deployment of Smart Applications

AmI research focuses on the improvement of human interactions with smart applications [12]. These improvements are made possible by the proposal of frameworks and platforms that facilitate the development of context-aware and dynamic applications. These platforms offer mechanisms to build such applications by handling data and events [16, 19] or by wrapping hardware and software capabilities into agents [9, 13]. However, it is often assumed that an underlying interoperable hardware and energy infrastructure already exists [24]. Meanwhile, the Internet of Things (IoT) aims to provide a global infrastructure for the information society, enabling advanced services by interconnecting physical and virtual "things" based on existing and evolving interoperable information and communication technologies [18]. The main challenge of the IoT is to achieve full interoperability of interconnected devices while guaranteeing the trust, privacy and security of communications [4]. However, a gap exists between AmI and the IoT. Indeed, because of the heterogeneity of such systems, it is difficult to have horizontal communication between connected devices. Present applications use devices that are vertically connected, from the device to an external server that collects and processes the data. The available commercial products are usually not directly interoperable. Moreover, this approach raises privacy questions: the

user does not own his data any more and privacy cannot be guaranteed that way. Hence, to fill this gap between IoT and AmI applications, adequate deployment mechanisms are required. We addressed the deployment problem in [25] by proposing to model the available hardware infrastructure and the needs of the applications using graphs that describe the various entities, their relations and properties. For deploying an application on the infrastructure, we proposed an extended graph matching algorithm for finding the hardware entities of the infrastructure that fulfil the requirements of the distributed application. However, this solution was centralised, which makes it unsuitable for real systems that need to take into consideration, among others, privacy and scalability. To address these issues, we propose a multi-agent-based distributed deployment software. Through its modularity, the multi-agent paradigm facilitates the local processing of data and guarantees the autonomy of the different parts of the hardware infrastructure, thus enhancing the privacy and robustness of the software.

This paper is organised as follows. Section 2 presents a scenario that illustrates the deployment of applications. Then, in Sec. 3 we explain why multi-agent systems are well-adapted to design the deployment software and ensure privacy and we detail this multi-agent structure. Section 4 presents our implementation using a goal-oriented approach. Section 5 shows similar works that use agents for the deployment of applications and privacy management. We conclude by presenting the next steps of this work.

2 Scenario

The scenario we use in this paper highlights both the dynamic deployment of distributed applications and the privacy management encapsulated in both agents and agent organisations. Mr Snow uses a *video doorkeeper* for dependant persons (e.g. visually impaired) application in his home. When someone rings at the door, the image of the entrance camera is displayed on a screen near Mr Snow, making sure he can properly see the person. He can then discuss with the person and decide whether or not to remotely open the door.

It is Saturday morning and Mr Snow is waiting for a parcel that will be delivered to his home at any time. While he is grooming himself in the bathroom, his neighbour, Mr Den, rings the door. The smart house, aware that Mr Snow is in his bathroom, selects the connected mirror of the bathroom, instead of any of the other display screens of the house, as a support to display the image stream of the entrance camera. Mr Snow, not being able to receive his guest, informs him, thanks to the microphone in the mirror, that he will meet him in an hour. After getting ready, Mr Snow goes to his neighbour. In the middle of their conversation, he is notified that an unkonwn man rings at his door again. He tries to recognise with his neighbour by displaying the image on Mr Den's television. By default, Mr Snow does not have the right to use any devices that he does not own, but the Mr Den has authorised him to access the television when he is at home. The doorkeeper application is redeployed dynamically to

use the requested hardware entities. Neither Mr Snow nor his neighbour know the visitor. Mr Snow decides to activate the microphone of the camera which allows him to learn that the unknown person is the expected transporter, which he can now go and see in person.

The important point in this scenario is not the video doorkeeper application, but the way it is deployed dynamically in the environment, considering the user's context. The scenario shows two deployment situations: (1) the application was deployed for use in the user's own home infrastructure, but in a less usual place: the bathroom; (2) the application was deployed on the infrastructure of another user, as the necessary access rights had been granted.

To achieve the deployment of any applications on an existing hardware infrastructure and to handle privacy constraints, we use the MAS features. The next section presents why MAS paradigm is suitable for our purpose.

3 Multi-Agent Modelling

Our scenario highlights several necessary specificities of the deployment software. This software has to dynamically deploy and undeploy distributed AmI applications in an environment that is also dynamic: when a visitor rings the doorbell, the deployment of the video doorkeeper should start, considering the available hardware entities and the location of the user, in order to choose the most relevant screen for displaying the image of the camera. Given its the distribution and openness that characterize the AmI domain, privacy is a very important characteristic of the deployment software. Privacy is defined by Alan Westin [32] as the claim of individuals, groups or institutions to determine for themselves when, how and to what extent information about them is communicated. In this scenario, we focus on *resource privacy*. Mr Snow is the owner of the hardware entities in his house and he does not want that unauthorised persons use or even know of the existence of these resources. At last, autonomy and robustness of the system are also very important specificities: if my neighbour's system failed, mine should continue to work normally and should not be impacted.

As the required software demands distribution, privacy, context management, autonomy and robustness, we identified MAS as a suitable solution. Through its modularity, this paradigm facilitates a local processing of the data and guarantees the autonomy of the different parts of the hardware infrastructure, thus handling aspects of privacy and robustness. To solve the dynamic deployment problem, we use the graph representation for the hardware infrastructure from our previous work [25]. Nodes represent hardware entities or relations between these entities and properties can be attached to each node. The requirements of the deployable applications are also described using such graphs. A graph matching algorithm can then be used on the available infrastructure graph to find the entities that can support the running of the application.

In the next sub-sections, we present the modelling of agents and the agent organisation for our deployment solution, while focusing on the encapsulation of resource privacy.

3.1 Agents and Artifacts

The deployment software involves the user deploying applications on an infrastructure. Three types of agent were therefore defined to represent and clearly separate each of the parties in handling the deployment: *User Agent*, *Application Agent* and *Infrastructure Agent*. A fourth type of agent was introduced for enhancing resource privacy: the *Infrastructure Super Agent*. For each type of agent we identified the main goals, that will then be described in Sec. 4:

- An *Infrastructure Agent* deals with a part of the global hardware infrastructure. It uses the graph representation of this available infrastructure [25] (hardware entities, relations and properties). This graph representation is never shared with other agents. The *Infrastructure Agent* reasons on it to propose partial solutions for the deployment of applications, thanks to a graph-matching algorithm. This class of agent has several goals, as it has to: (1) keep the infrastructure graph up to date; (2) propose solutions for the deployment of applications, considering the available hardware infrastructure, but also the sharing and privacy policy and (3) deploy or (4) undeploy functionalities of an application.
- An *Infrastructure Super Agent* is a representative of a set of *Infrastructure Agents* which are related to it forming a *group*. It acts as a proxy between the agents inside and outside of the group.
- An *Application Agent* manages an entire application during its runtime. It has a graph-based description of the application [25]. An example of such graph is represented in Fig. 1: the upper part represents the functionalities of the application and the bottom part shows their hardware requirements. The main goals of this class of agent are: (1) guarantee the consistency of the application and (2) deploy or (3) undeploy functionalities of the application if necessary. The *Application Agent* has to interact with several *Infrastructure Agents* in order to deploy the functionalities of the application over the infrastructure.
- At last, the *User Agent* is the interface between the user and the other agents of the deployment software. Through this agent, a user can request the (1) deployment or (2) undeployment of applications.

In addition to these four classes of agent, we also propose two classes of artifact which are resources and tools that can be instantiated and/or used by agents in order to interact with the environment [27]:

- *Deployment artifacts* [14] can be used by the *Infrastructure Agents* in order to effectively deploy some parts of an application, or configure hardware entities so that they can be used by the application.
- The second class of artifact are the *functionalities* of the applications themselves. Some of them can provide useful contextual information to the deployment software (location of a user, available bandwidth, ...), to help the agents keeping their application or infrastructure graph up to date.

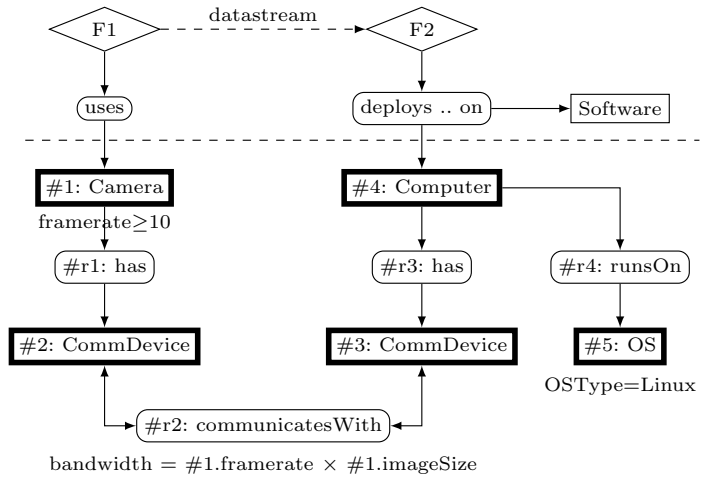


Fig. 1. Example of a basic application graph

In the video doorkeeper scenario, there are three *Infrastructure Agents*. The first one manages the hardware entities located in the living room of Mr Snow, like the television. The second one manages the entities of the bathroom like the connected mirror. And the last one manages the house of the neighbour. We also find two *Application Agents*. The first one manages the video doorkeeper application; when a visitor rings the doorbell, this *Application Agent* triggers the deployment of the video interaction functionality. The second one manages the application which provides the location of the Mr Snow inside his own house to his own *Infrastructure Agents*. The contextual location information is useful for deploying other applications. Indeed, the display screen of the video doorkeeper application has to be chosen near the user. Then, we have two *User Agents*. The first one is the interface between the deployment software and Mr Snow, and the second one is owned by Mr Snow's neighbour. At last, we have a certain number of deployment artifacts that can configure the display screens, the cameras, or deploy software on devices (TV box, connected mirror etc.).

The agent decomposition encapsulates a part of the privacy mechanism. Indeed, the graph representation of the available hardware infrastructure managed by an *Infrastructure Agent* is only known by this agent and is never shared with others. Moreover, the architecture used helps keeping a clear separation between the applicative part, managed by the *Application Agents*, and the hardware part, monitored by the *Infrastructure Agents*. As agents only have a local view of the system, the privacy is enhanced. In the next sub-section, we show how using this agent organisation improves resource privacy and allows the definition of privacy policies for the use of these resources.

3.2 Organisation and Interactions

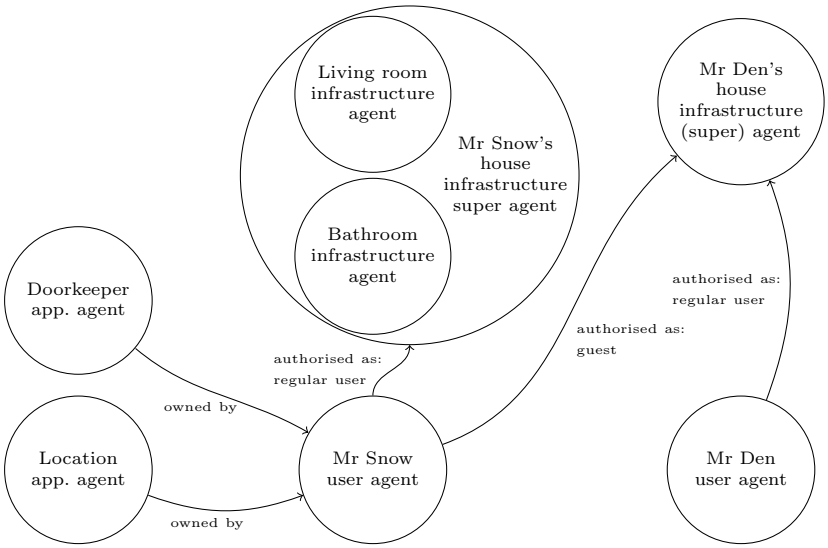


Fig. 2. Agent organisation

The agents presented in the previous section are constrained by the organisational structure and follow the privacy policy given by the owner. *Infrastructure Agents* can be grouped behind an *Infrastructure Super Agent* which, as stated before, acts as a proxy for the agents of the group. From an outside view, this *Infrastructure Super Agent* is seen as a normal *Infrastructure Agent*.

In our scenario, the living room and the bathroom *Infrastructure Agents* of Mr Snow are grouped behind an *Infrastructure Super Agent* representing the house of Mr Snow. Similarly, the *Infrastructure Agent* managing the house of Mr Snow's neighbour is a super agent, regrouping several *Infrastructure Agents* (or other sub-super agents). The advantage of such organisation is that it is easy to abstract groups of agents and make them invisible from the outside, resulting in a multi-scale organisation that helps improve privacy. Indeed, Mr Snow knows about his own *Infrastructure Agents* (bathroom and living room), but he does not have to know anything about the details of Mr Den's infrastructure organisation. If he wants to interact with his neighbour's house, he has to interact with Mr Den's *Infrastructure Super Agent* with the required access rights granted (as described below). The upper part of Fig. 2 shows the organisation of the *Infrastructure Agent* from Mr Snow's point of view. This kind of organisation can be implemented with both a hierarchy or a holarchy [17]. In a hierarchical organisation [15], *Super Agents* are represented by a software agent that acts as a proxy between the agents of the group and the outside. This agent is the

favoured interface between the group and the outside. In a holarchy [20], a super agent is not a concrete software agent but is represented by the sum of all *Infrastructure Agents* of the group. Each of these agents can be a representative of the super agent, unlike in a hierarchy where there is only one representative that may become a single point of failure. A holarchy is more complex to implement, but it is also more flexible and can evolve dynamically. Some sophisticated hierarchies are similar to holonic organisations [17]. In our implementation, we used hierarchies because they are easier to implement and debug, but both models could have been implemented.

The organisation of *Infrastructure Agents* ensures privacy by hiding information about the structure of its sub-organisations. However, to improve privacy by controlling the use of resources, we also propose sharing policies. *User Agents* can be authorised, by the owner of some hardware infrastructure, to use some parts of its infrastructure, and cooperate with the associated *Infrastructure Agents* or *Super Agents*, to deploy applications. If a *User Agent* is not authorised by the *Infrastructure (Super) Agent*, it cannot use the hardware resources proposed by this agent. Otherwise, it can have different authorisation levels. For example: (1) Administrator level: the agent (and implicitly its user) has full access to the resources proposed by the *Infrastructure Agent*, can reconfigure the *Super Agent* organisation and manage the authorisation levels; (2) Regular user: the agent has access to the resources of the *Infrastructure (Super) Agent* but it cannot reconfigure authorisation levels or agent organisation and (3) Guest: the agent has a restricted access to the resources. Only the resources considered as non critical by an administrator are allowed to be shared. These authorisation levels are not limited to three and can be modified by the administrator of the *Super Agent*. In the video doorkeeper scenario, Mr Snow's *User Agent* is a Regular user for his home *Infrastructure Super Agent*, but it is just a Guest to his neighbour's home *Infrastructure Super Agent*. As such, it has only access to the television of Mr Snow's neighbour. This allows to ensure privacy of the other resources of Mr Den. The *Application Agents* have the same authorisation level as the *User Agent* that creates them. They can interact with the authorised *Infrastructure Agents* in order to effectively deploy their application. Figure 2 shows the agent structure of the doorkeeper scenario; the agent organisation, the authorisation level, and the *Application Agents* that are bounded to their *User Agent* creator.

In this section, we have shown how privacy is preserved through encapsulation in our MAS. *Infrastructure Agents* keep the information about the hardware infrastructure secret. The *Infrastructure Agent* hierarchy keeps the details of the agent organisation hidden. Privacy policies can allow or prevent the sharing of resources to *User Agents*. This results in privacy by design. In the next section, we present how we designed and implemented the MAS using a goal-driven approach.

4 Design and Implementation

The agents were designed using a goal-based model due to its benefits to the autonomy and robustness of the application. Goals are specified by describing their associated plans: higher level *goal plans* describing relationships between goals and lower level *action plans* for concrete actions. This goal-based representation is based on the Goal-Plan Separation (GPS) approach [8], where each agent has a main goal plan (i.e. plan without any actions, so only decisions, perceptions and goal adoptions) that describes the top level behaviour, which can be pursued using other goal plans or directly action plans (i.e. plan without any goal adoptions). This approach helps handle agent complexity through a multi-level description, from top level abstract behaviours with goals to concrete action plans. Using goal-plans also has the advantage of specifying the relationships between goals in a plan format.

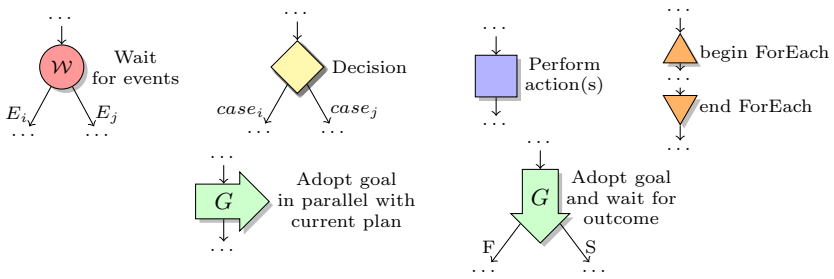


Fig. 3. Flowchart nodes for efficiently describing the plans of goal-driven agents

Plans are represented using a flowchart notation we adapted for modelling goal-driven agents (Fig. 3). The notation contains the main elements that allow for the behaviours of agents to be defined: decisions, event perceptions (wait), actions and goal adoptions. Parallel executions are launched when adopting goals. For this application, we considered a simple goal model (similar to a *perform* goal [7]) where a goal is successful ("S") when the plan executing for it ends with "End ok". This allowed us to keep a simple goal life-cycle appropriate for using in our application, while still benefiting from the features of the goal-based design.

We continue by describing in detail the agents of the system. Since the *Infrastructure Super Agent* is only a proxy between the agents of the group it represents and the other agents outside this group, its implementation is not detailed here. In what follows, P_{Xi-j} are the plans for a goal G_{Xi}.

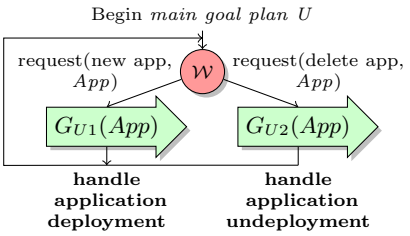


Fig. 4. *User Agent*: main goal plan

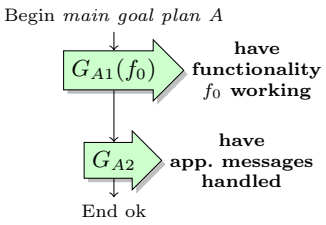


Fig. 5. *Application Agent*: main goal plan

4.1 User Agent

The *User Agent* acts as an interface between the user and the deployment MAS. The goal plan of the *User Agent* (Fig. 4) waits for user input and, depending on the received request, adopts the necessary goal, corresponding to the agent functions identified in Sec. 3.1. The plans of G_{U2} and G_{U1} are similar: they create an *Application Agent* or request an application to be undeployed, wait for a confirmation and display the information to the user. The *User Agent* also allows the changing of the privacy policies, but this was not represented here.

4.2 Application Agent

The *Application Agent* is created by a *User Agent*. It tries to deploy a precise application by cooperating with one or more known *Infrastructure (Super) Agents*, from which it does not need to have any infrastructure details.

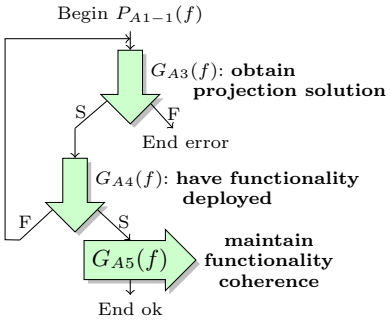


Fig. 6. *Application Agent*: goal plan for G_{A1}: "have functionality f working"

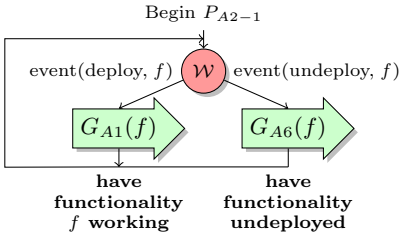


Fig. 7. *Application Agent*: goal plan for G_{A2}: "have app. messages handled"

Upon its creation, an *Application Agent* adopts two goals (Fig. 5): G_{A1} for deploying an initial functionality (Fig. 6) and G_{A2} that waits for internal events for new deployments or undeployments (Fig. 7). The deployment is done in two

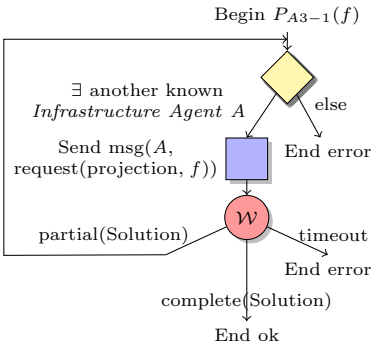


Fig. 8. *Application Agent*: plan for G_{A3}: "obtain projection solution"

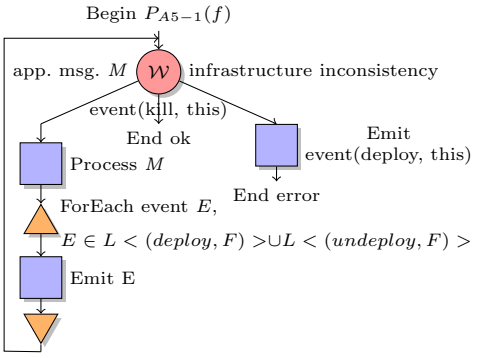


Fig. 9. *Application Agent*: plan for G_{A5}: "maintain functionality coherence"

steps: first the agent obtains a deployment solution from *Infrastructure Agents* via G_{A3} and then it requests the deployment according to this solution through G_{A4}. The *Application Agent* sends a list of the requirements described in the application graph to the *Infrastructure Agent* and the solution it receives contains the list of requirements that could be fulfilled. Note that the reply does not contain any actual infrastructure details, which is important for the privacy of the infrastructure. It can be seen (Fig. 8) that the agent may need to call multiple *Infrastructure Agents* in order to obtain a complete deployment solution. Indeed an *Infrastructure Agent* tries to find in its own infrastructure the hardware entities that match the requirements of the application. However, if these requirements only partially match, the *Infrastructure Agent* will return a partial solution to the *Application Agent*. In this case, the latter will call another *Infrastructure Agent* that will continue to match the requirements of the application. Once a solution has been found, the *Application Agent* interacts again with the concerned *Infrastructure Agents* to effectively deploy the functionalities of the application: plan P_{A4-1} simply sends a message and waits for a confirmation.

After a functionality was deployed, the agent monitors it through G_{A5} (with its plan in Fig. 9) in order to adapt the deployment to the current context: infrastructure inconsistency (e.g. changing infrastructure availability, changing user location) and messages from the application itself (e.g. new guest at the door). An application message can result in multiple requests for deployments and undeployments. Internal events are used to control the execution of different plans. Deploy and undeploy events originate in the plan for G_{A5} and trigger the adoption of G_{A1} for the deployment of other functionalities or redeployment of the current one, and G_{A6} for the undeployment of the functionality. As each functionality is monitored by an instance of G_{A5}, in case of an undeployment, the plan of G_{A6} signals the corresponding G_{A5} to stop through a *kill* event (besides sending a request message to the corresponding *Infrastructure Agent*).

Note here that the *Application Agents* only handle the application deployment. The application itself is in charge of its own actions, data and privacy.

4.3 Infrastructure Agent

An *Infrastructure Agent* receives requests from *Application Agents* that it tries to satisfy (Fig. 10). Only requests originating from known *User Agents* are treated, in other words only applications from agents that were granted one of the levels of authorisation are accepted.

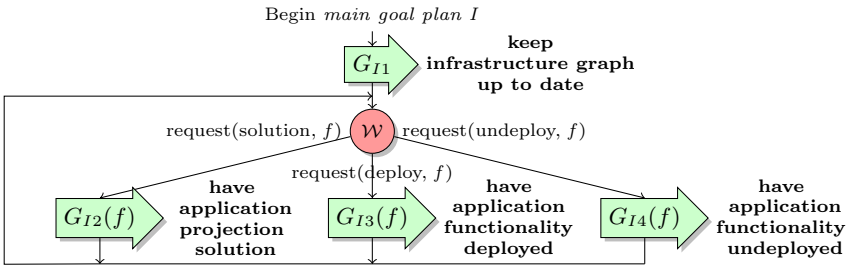


Fig. 10. *Infrastructure Agent*: main goal plan

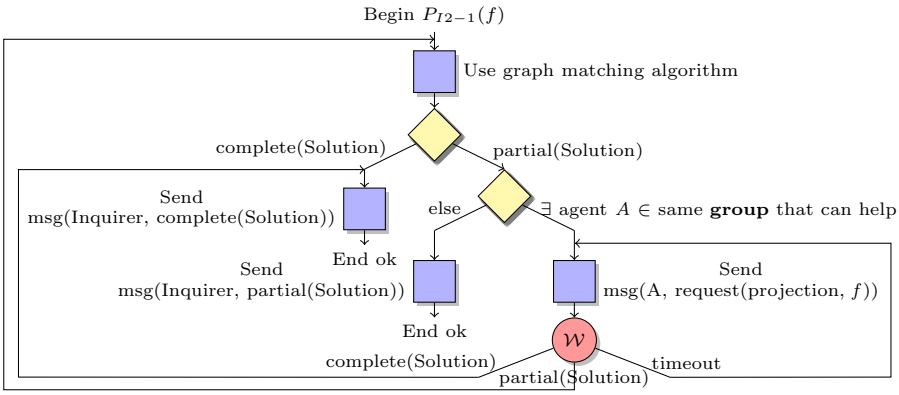


Fig. 11. *Infrastructure Agent*: plan for G_{I2}: "have application projection solution". "Inquirer" can be an *Application Agent* or another *Infrastructure Agent*.

When it receives a request for a deployment solution, the *Infrastructure Agent* uses the graph matching algorithm to determine if it can fulfil the requirements of the request (Fig. 11) using the devices it manages. The algorithm takes into

consideration the levels of authorisation of the involved *User Agents*. If it cannot produce a complete solution, the *Infrastructure Agent* requests the help of other agents in its group, but without informing the *Application Agents*. In this way, the components of the infrastructure remain private. If a complete solution is eventually produced and the *Infrastructure Agent* is given the order to deploy the application, it will dispatch the deployment tasks to its own deployment artifacts as well as to any other *Infrastructure Agents* that were included in the final solution. In case any of these requests fails (e.g. an artifact malfunctions), the whole application is undeployed and the *Application Agent* is informed, which will cause it to restart the deployment procedure.

In parallel with the request handling, the agent also adopts G_{I1} which listens for agent and artifact information in order to manage the graph the devices corresponding to the *Infrastructure Agent*. In case of an inconsistency (e.g. Mr Snow leaves Mr Den's home, so any display he used there are no longer relevant for the application), the agent informs the *Application Agents* that it will need to redeploy the concerned parts of their applications.

4.4 Implementation

A demonstration model of the deployment software has been developed in an apartment replica attached to our laboratory. This home replica implements various scenarios applied to home care for dependent persons, including the presented scenario. These scenarios are using commercial connected devices tweaked to be horizontally connected, thanks to the deployment software. These demonstrations run continuously and can be tested by any visitor.

Our goal is to run the MAS on different devices like smartphones or embedded systems with few resources. Most of existing regular MAS platforms like Jade for instance are memory-consuming and Java-oriented platforms [21]. They are not suitable for our purpose. That is why we designed our own MAS platform in JavaScript. Indeed, web technologies are fully interoperable and the agents can easily be run on devices like smartphones or the Raspberry Pi. Visualisation and interfaces are also JavaScript web applications. The agents embed a monitoring and debugging web server that proposes interfaces for interacting with it. The effective deployment is handled by deployment artifacts. The demonstration model handles *ssh* and *puppet* artifacts in order to deploy and run software on UNIX systems (computers, micro-computers, Unix-based devices etc.). We also implemented a specific deployment artifact that configures the frame rate of IP cameras. In this implementation we mostly used IP devices. We also integrated EnOcean devices. These devices, however, are handled by a hard-coded gateway that extends the IP network to EnOcean devices. Next stage will be to handle multiple means of communication by automatically deploying gateways or proxies between the devices when needed. At last, the agent implementation was, in first place, not obvious. The multi-level GPS approach made it intuitive to develop.

This realisation helps us to figure out the difficulties of handling the heterogeneity of hardware entities. We are now able to handle applications through an

AppStore for Smart Homes. These applications can be automatically deployed in a real environment, using the available hardware devices, and including mechanisms to ensure privacy management of the resources. This provides a concrete base for the implementation of a complete middleware for the deployment of distributed applications in a smart environment.

5 Related Work

Several works address the deployment problem. Braubach and al. [6] propose a deployment reference model based on a MAS architecture (e.g. agent services) for deploying MAS applications. As an agent is a software entity, the deployment of agents does not have to deal with the high heterogeneity of hardware entities. Some other works in the service-oriented architectures (SOA) community [3] reason on deployment patterns, that specify the structure and constraints of composite solutions on the infrastructure, in order to compose services. Contrary to our approach, the cited paper refers not to the localisation of resources and installation of software, but rather to the binding of existing resources in order to provide the desired composition of services. This is realised using a centralised graph-matching algorithm that takes into account the various requirements for the given service. Flissi and al. [14] propose a meta-model for abstracting the concepts of the deployment of software over a grid. All these works have shortcomings when considering their use for deploying AmI applications on the IoT infrastructure. Some do not take into consideration the heterogeneity of the hardware and software, as well as the interaction between the two layers (i.e. software and hardware). Others do not tackle the privacy problem. And some propose centralised solutions that are not scalable for real life AmI applications. Our MAS approach takes these problems into consideration: scalability is handled thanks to the agent structure; the autonomy of agents, organisation and privacy policies provide resource privacy; and heterogeneity is supported by the description of the system and the reasoning mechanisms that find projections of applications on the infrastructure.

Privacy in multi-agent systems has already been well explored. Such and al. [29] categorise research on data privacy on different levels: collection, disclosure, processing and dissemination. Multi-agent system specificities have been used to propose different manners of handling the data privacy. Some works focus on norms [5, 22] and privacy policies [30, 11, 31], checked by agent brokers to control the disclosure of the data. Other works [26, 28] use social relationships like trust, intimacy or reputation to select the agents with which data can be shared. Trusted third parties are already used in [23, 1, 10] in order to anonymise the data or the metadata (e.g. IP address, receiver or sender identity), and also to check disclosure authorisations. At last, some works [2] focus on integrating secure communication in the agent platforms by using well known encryption protocols. All these works use MAS in order to provide data privacy. In our work, as explained in Sec. 3, we merely take advantage of MAS properties to handle the privacy of the hardware resources and of the structure of the system.

6 Conclusion and Future Work

In this paper, we presented a multi-agent solution for reasoning on the dynamic deployment of distributed applications in ambient systems. We described the modelling of the system and presented the specifications of the goal-based agents. We illustrated the MAS using a context-aware video doorkeeper scenario. In this scenario, a doorkeeper application is dynamically deployed in order to route the video stream of the entrance hall camera to a relevant screen, near the user, thanks to contextual information about his location. Other scenarios are possible using the apartment replica we used.

The MAS proposed in this paper contains four classes of goal-directed agent to handle a clear separation between the hardware and software layers and to ensure resource privacy in ambient systems. In order to preserve the privacy of the resources, the graph models of the infrastructure are handled locally by the concerned agents. The matching between the requirements of the application functionalities and the hardware entities is performed using a decentralised version of an existing graph-matching algorithm adapted to the graph formalism we use.

The use of MAS made it possible to introduce privacy measures at architecture and organisation level, on top of which we were able to add a user-defined privacy policy mechanism. This was an important criterion for the choice of the agent paradigm since in the domain of Ambient Intelligence there are often different infrastructure owners that need to ensure the privacy of their resources. The separation between the applicative and the infrastructure layers, together with the decentralised approach also enhance the robustness of the solution. The clearly delimited entities, with either virtual (the applications) or physical (users, infrastructure elements) correspondents, guided the agentification. The use of a goal-based representation for agents together with the Goal-Plan Separation approach facilitated the modelling task. The specific plan notation was efficient in describing the agent plans both during design and for presentation purposes.

In terms of future work, for the deployment software, data privacy in the deployed applications should also be taken into consideration in addition to the resource privacy discussed here. We would like to facilitate the local processing and storage of the data by defining data privacy policies which should be facilitated by the modularity of the MAS. The user should decide which kind of data he authorises to come out of his home infrastructure. This would impact the reasoning on the deployment: the hardware entities would have to be filtered with respect to this new data privacy policy. In the interest of the engineering of multi-agent systems, we are studying the goal-based modelling approach with GPS agents and the plan notation for the extension towards a development methodology for robust software.

References

1. Aïmeur, E., Brassard, G., Fernandez, J.M., Onana, F.S.M.: Privacy-preserving demographic filtering. In: Proceedings of the 2006 ACM Symposium on Applied Computing. pp. 872–878. SAC '06, ACM, New York, NY, USA (2006)
2. Alberola, J., Such, J., Garcia-Fornes, A., Espinosa, A., Botti, V.: A performance evaluation of three multiagent platforms. Artificial Intelligence Review 34(2), 145–176 (2010)
3. Arnold, W., Eilam, T., Kalantar, M., Konstantinou, A., Totok, A.: Automatic realization of soa deployment patterns in distributed environments. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) Service-Oriented Computing ICSOC 2008, Lecture Notes in Computer Science, vol. 5364, pp. 162–179. Springer Berlin Heidelberg (2008)
4. Atzori, L., Iera, A., Morabito, G.: The internet of things: A survey. Computer Networks 54(15), 2787 – 2805 (2010)
5. Barth, A., Datta, A., Mitchell, J., Nissenbaum, H.: Privacy and contextual integrity: framework and applications. In: Security and Privacy, 2006 IEEE Symposium on. pp. 15 pp.–198 (May 2006)
6. Braubach, L., Pokahr, A., Bade, D., Krempels, K.H., Lamersdorf, W.: Deployment of distributed multi-agent systems. In: Gleizes, M.P., Omicini, A., Zambonelli, F. (eds.) Engineering Societies in the Agents World V, Lecture Notes in Computer Science, vol. 3451, pp. 261–276. Springer Berlin Heidelberg (2005)
7. Braubach, L., Pokahr, A., Moldt, D., Lamersdorf, W.: Goal Representation for BDI Agent Systems. In: Bordini, R., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.) Programming Multi-Agent Systems, LNCS, vol. 3346, pp. 44–65. Springer Berlin Heidelberg (2005)
8. Caval, C., El Fallah Seghrouchni, A., Taillibert, P.: Keeping a clear separation between goals and plans. In: Dalpiaz, F., Dix, J., van Riemsdijk, M. (eds.) Engineering Multi-Agent Systems, Lecture Notes in Computer Science, vol. 8758, pp. 15–39. Springer International Publishing (2014)
9. Chen, H., Finin, T.W., Joshi, A., Kagal, L., Perich, F., 0001, D.C.: Intelligent agents meet the semantic web in smart spaces. IEEE Internet Computing 8(6), 69–79 (2004)
10. Cissée, R., Albayrak, S.: An agent-based approach for privacy-preserving recommender systems. In: Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems. pp. 182:1–182:8. AAMAS '07, ACM, New York, NY, USA (2007)
11. Crépin, L., Demazeau, Y., Boissier, O., Jacquenet, F.: Sensitive Data Transaction in Hippocratic Multi-Agent Systems. In: Engineering Societies in the Agents World IX, pp. 85–101. Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2009)
12. Ducatel, K., Bogdanowicz, M., Scapolo, F., Leijten, J., Burgelman, J.: Scenarios for ambient intelligence in 2010 (2001)
13. El Fallah Seghrouchni, A., Olaru, A., Nguyen, N.T.T., Salomone, D.: Ao dai: Agent oriented design for ambient intelligence. In: Desai, N., Liu, A., Winikoff, M. (eds.) PRIMA. Lecture Notes in Computer Science, vol. 7057, pp. 259–269. Springer (2010)
14. Flissi, A., Dubus, J., Dolet, N., Merle, P.: Deploying on the grid with deployware. In: Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium on. pp. 177–184 (May 2008)

15. Fox, M.S.: An organizational view of distributed systems. IEEE Transactions on Systems, Man, and Cybernetics 11(1), 70 – 80 (1981)
16. Hellenschmidt, M., Kirste, T.: A generic topology for ambient intelligence. In: Markopoulos, P., Eggen, B., Aarts, E.H.L., Crowley, J.L. (eds.) EUSAI. Lecture Notes in Computer Science, vol. 3295, pp. 112–123. Springer (2004)
17. Horling, B., Lesser, V.: A survey of multi-agent organizational paradigms. Knowl. Eng. Rev. 19(4), 281–316 (Dec 2004)
18. ITU-T: Overview of the internet of things, recommendations (2012)
19. Johanson, B., Fox, A., Winograd, T.: The interactive workspaces project: Experiences with ubiquitous computing rooms. IEEE Pervasive Computing 1(2) (2002)
20. Koestler, A.: The Ghost in the Machine. Hutchinson (1967)
21. Kravari, K., Bassiliades, N.: A survey of agent platforms. Journal of Artificial Societies and Social Simulation 18(1), 11 (2015)
22. Krupa, Y., Vercouter, L.: Contextual integrity and privacy enforcing norms for virtual communities. In: Boissier, O., El Fallah Seghrouchni, A., Hassas, S., Maudet, N. (eds.) MALLOW. CEUR Workshop Proceedings, vol. 627. CEUR-WS.org (2010)
23. Menczer, F., Street, W., Vishwakarma, N., Monge, A., Jakobsson, M.: IntelliShopper: A proactive, personal, private shopping assistant. In: Proc. 1st ACM Int. Joint Conf. on Autonomous Agents and MultiAgent Systems (AAMAS) (2002)
24. O'Hare, G.M.P., Collier, R., Dragone, M., O'Grady, M.J., Muldoon, C., de J. Montoya, A.: Embedding agents within ambient intelligent applications. In: Bosse, T. (ed.) Agents and Ambient Intelligence, Ambient Intelligence and Smart Environments, vol. 12, pp. 119–133. IOS Press (2012)
25. Piette, F., Dinont, C., El Fallah Seghrouchni, A., Taillibert, P.: Deployment and configuration of applications for ambient systems. Procedia Computer Science 52, 373 – 380 (2015), the 6th International Conference on Ambient Systems, Networks and Technologies (ANT-2015)
26. Ramchurn, S.D., Huynh, D., Jennings, N.R.: Trust in multi-agent systems. Knowl. Eng. Rev. 19(1), 1–25 (Mar 2004)
27. Ricci, A.: Agents and coordination artifacts for feature engineering. In: Ryan, M.D., Meyer, J.J.C., Ehrich, H.D. (eds.) Objects, Agents, and Features. Lecture Notes in Computer Science, vol. 2975, pp. 209–226. Springer (2003)
28. Such, J.M., Espinosa, A., GarcíA-Fornes, A., Sierra, C.: Self-disclosure decision making based on intimacy and privacy. Inf. Sci. 211, 93–111 (Nov 2012)
29. Such, J.M., Espinosa, A., Garca-Fornes, A.: A survey of privacy in multi-agent systems. The Knowledge Engineering Review 29, 314–344 (Mar 2014)
30. Tentori, M., Favela, J., Rodriguez, M.D.: Privacy-aware autonomous agents for pervasive healthcare. IEEE Intelligent Systems 21(6), 55–62 (Nov 2006)
31. Udupi, Y.B., Singh, M.P.: Agents and peer-to-peer computing. chap. Information Sharing Among Autonomous Agents in Referral Networks, pp. 13–26. Springer-Verlag (2010)
32. Westin, A.F.: Privacy and Freedom. Atheneum, New York (1967)

How testable are BDI agents? An analysis of branch coverage

Michael Winikoff[1]

University of Otago, Dunedin, New Zealand.

Abstract. Before deploying a software system, it is important to assure that it will function correctly. Traditionally, this assurance is obtained by testing the system with a collection of test cases. However, since agent systems exhibit complex behaviour, it is not clear whether testing is even feasible. In this paper we extend our understanding of the feasibility of testing BDI agent programs by analysing their testability with respect to the *all edges* test adequacy criterion, and comparing with previous work that considered the *all paths* criterion. Our findings include that the number of tests required with respect to the all edges criterion is much lower than for the all paths criterion. We also compare BDI program testability with testability of (abstract) procedural programs.

1 Introduction

When any software system is deployed, it is important to have assurance that it will function as required. Traditionally, this assurance, encompassing both validation and verification[1], is obtained by testing, and there has been work on tools and techniques for testing agent-based systems (e.g. [9, 11, 14, 15, 24]). However, there is a general intuition that agents exhibit behaviour that is complex. More precisely, due to the need to handle dynamic and challenging environments, agents need to be able to achieve their objectives flexibly and robustly, which requires richer and more complex possible behaviours than traditional software. Therefore, a key question is *whether agent systems are harder, and possibly even infeasible, to assure by testing*.

Before proceeding further we need to define what we mean by a program being testable. Rather than define testability as a binary property, we define it as a numerical measure of the effort required to test a program[2]. Specifically, given a program and a *test adequacy criterion* [13], we consider the testability of a program to be the smallest number of tests that would be required to satisfy the criterion. For example, given the (very simple!) program "**if** c **then** s_1 **else** s_2", then we need two tests to cover all edges in the control-flow graph corresponding to this program, which satisfies the "all edges" test adequacy criterion (defined below),

[1] More precisely: "software quality assurance (SQA) is a set of activities that define and assess the adequacy of software processes to provide evidence that establishes confidence that the software processes are appropriate and produce software products of suitable quality for their intended purposes." (ISO/IEC TR 19759:2015(E), page 10-5)

[2] We focus on system testing. See [20, Section 7] for a discussion of different forms of testing.

The *all paths* and *all edges* test adequacy criteria are defined with respect to a control-flow graph. A given program P corresponds to a graph where nodes are statements (or, for agents, actions), and edges depict the flow of control: a node with multiple outgoing edges corresponds to a choice in the program. A single test corresponds to a path through the program's control-flow graph from its starting node to its final node (we assume that there is a unique start node S and a unique end node E, which can be easily ensured). The *all paths* criterion is satisfied iff the set of tests in the test suite T cover all *paths* in the control flow graph. The *all edges* criterion is satisfied iff the set of paths in the test suite T covers all *edges* in the control-flow graph [13]. The all edges criterion is also referred to as "branch coverage".

Given the importance of assurance, and the focus on testing as a means of obtaining assurance[3], there has been surprisingly little work that has considered whether testing agent systems is even feasible. In fact, the only work that we are aware of that considers this question is the recent work by myself & Cranefield[4] [20, 21], which investigated the testability of Belief-Desire-Intention (BDI) agent programs with respect to the *all paths* test adequacy criterion. Winikoff & Cranefield concluded that BDI agent programs do indeed give rise to a very large number of possible paths (see left part of Table 1), and therefore they concluded that whole BDI programs are likely to be infeasible to assure via testing[5]. However, they do acknowledge that the all paths criterion is known to be overly conservative, i.e. it requires a very large number of tests. Specifically, all paths *subsumes* a wide range of other criteria, including all edges (e.g. see Figure 7 of Zhu *et al.* [25] and Figure 6.11 (page 480) of Mathur [13]). This means that the question of whether (whole) BDI agent programs can be feasibly tested is still open. This paper aims to address this question by considering testability with respect to the *all edges* [13] test adequacy criterion. The all edges criterion is regarded as "*the generally accepted minimum*" [12]. In essence, previous work [20] has provided an *upper* bound ("if we use a strong criterion, then it's this hard"). This paper provides a *lower bound* ("if we use a weaker criterion, than it's this hard").

The remainder of this paper is structured as follows. We (briefly) review BDI agent programs in Section 2. Section 3 is the core of the paper: it derives equations that compute for a given BDI program P the number of tests that are required to satisfy the all edges criterion. We then use these equations to compare testability (with respect to all edges) with testability with respect to all paths (Section 4). We also compare all edges testability for BDI programs with all edges testability for (abstract) procedural programs, in order to answer the question of whether BDI programs are *harder* to test than procedural programs with respect to the all edges criterion (Section 5). Finally, in Section 6 we conclude.

[3] There is also a body of work on formal methods (primarily model checking) as a means of assurance [23, 3, 16, 6, 8, 10, 7]. However, despite considerable progress, these are not yet ready to handle realistic programs (e.g. see [8]).

[4] To avoid confusion between this paper and the earlier work, I will refer to my earlier work with Stephen Cranefield as "Winikoff & Cranefield" in the remainder of this paper.

[5] They also compared BDI programs with procedural programs, and found that BDI programs are *harder* to test than equivalently sized procedural programs, with respect to the all paths criterion.

2 Belief-Desire-Intention (BDI) Agents

The Belief-Desire-Intention (BDI) model [18, 4, 5] is widely-used, and is realised in many agent-oriented programming languages (AOPLs) (e.g., [1, 2]). It provides a human-inspired metaphor and mechanism for practical reasoning, in a way that is appropriate for achieving robust and flexible behaviour in dynamic environments.

A BDI agent program Π consists of a sequence of plans $\pi_1 \ldots \pi_n$ where each plan π_i consists of a triggering goal[6] g_i a context condition c_i and plan body b_i. The plan body is a sequence of steps $s^i_1 \ldots s^i_{m_i}$ with each step being either an action or a sub-goal.

Due to space limitations, we give an informal summary of the semantics. Formal semantics can be easily defined following (e.g.) [17, 22, 19]. These semantics are common to the family of BDI programming languages (e.g. PRS, dMARS, JAM, AgentSpeak, JACK). A BDI program's execution begins with a goal g being posted. The first step is to determine the subset of *relevant* plans $\Pi_R \subseteq \Pi$ which is those plans π_i where the plan's trigger g_i can be unified with g. The second step is to determine the subset of *applicable* plans $\Pi_A \subseteq \Pi_R$ which is those plans π_i where the plan's context condition c_i holds with respect to the agent's current beliefs. The third step is to select one of the applicable plans $\pi_j \in \Pi_A$. The body b_j of the selected plan π_j is then executed. The execution is done step-by-step, interleaved with further processing of goals (and belief updates as information from the environment is received).

An important aspect of BDI execution is failure handling. A step in a plan body can fail. For an action, this can be because the action's preconditions do not hold, or due to the action simply not proceeding as planned (the environment is not always benign!). For a sub-goal, failure occurs when there is no applicable plan. When a plan step fails, the execution of the sequence of steps is terminated, and the plan is deemed to have failed.

A common way of dealing with the failure of a plan π_i which was triggered by goal g is to *repost* the goal g, and select another plan instance. More precisely, Π_A is re-computed (since the agent's beliefs might have changed in the interim), but with π_i excluded. A plan (instance) that has failed cannot be selected again when its triggering goal is reposted.

For the purposes of the analysis of this paper we consider a BDI agent program to be defined by the grammar below. This grammar simplifies from real BDI agent programs in a number of ways. Firstly, instead of a plan body having sub-goals g, with the relevant and applicable plan sets being derived from the plan library Π, we instead associate with each (sub-)goal g a set of plans[7] denoted $g^{\mathcal{P}}$ (where $\mathcal{P}$ is a set of plan instances). Because we have done this, we do not need to represent the plan library: a BDI program is simply a single (possibly quite complex) expression in the grammar below. Secondly, we follow CAN [22] in using an auxiliary "backup plan" construct to capture failure handling. Finally, we elide conditions: since the all edges criterion

[6] For the purposes of this paper we ignore other possible plan triggers provided by some AOPLs, such as the addition/removal of belief, and the removal of goals.

[7] For the moment we avoid specifying whether $\mathcal{P}$ is the set of relevant plans or applicable plans. The analysis in the next section considers both cases.

considers control-flow, we do not need to model the conditions that are used to decide which edge to take in the control flow graph.

We therefore define a BDI program P using the grammar:

$$P ::= a \mid g^{\{P^*\}} \mid P_1;P_2 \mid P_1 \triangleright P_2$$

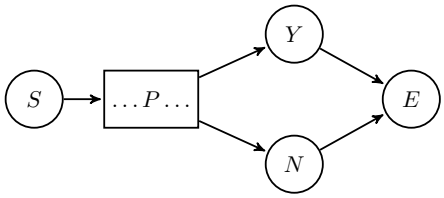
where a is an action (and we use $a_1, a_2, a_3, \ldots$ to distinguish actions), $g^{\mathcal{P}}$ is a (sub-)goal with associated plans $\mathcal{P} = \{P_1, \ldots, P_n\}$ (a set of plans), $P_1; P_2$ is a sequence, and $P_1 \triangleright P_2$ represents a "backup plan": if P_1 succeeds, then nothing else is done (i.e. P_2 is ignored), but if P_1 fails, then P_2 is used. Any BDI program with given top-level goal can be mapped into a BDI program in this grammar. Note that this grammar does not capture some of the constraints of BDI programs (e.g. that a goal cannot directly post a sub-goal).

3 All-Edge Coverage Analysis

This section is the core of the paper. It derives equations that answer the question: "how many test cases (paths) are required to cover all edges in the control-flow graph corresponding to a given BDI program?".

Recall that a BDI agent program P can be either an action a, a sub-goal $g^{\mathcal{P}}$, a sequence (";"), or an alternative ("$\triangleright$"). We consider each of these cases in turn. For each case we consider how the construct is mapped to a control-flow graph, and then how many paths are required to cover all edges in the graph.

One important feature of BDI programs is that the execution of a BDI program (or sub-program) can either succeed or fail. A failed execution triggers failure handling. We represent this by mapping a program P to a graph (see right) where there is a start node S, the program P is mapped to a graph that is reachable from S, and that has *two* outgoing edges: to Y (corresponding to a successful execution) and N (corresponding to a failed execution). There are edges from Y and N to the end node E.



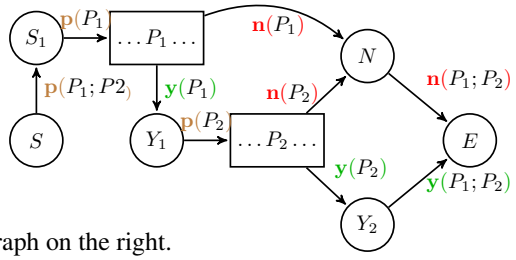
Note that there is an important difference between the notion of a test for a conventional program and for an agent system. In a conventional program a test corresponds to the setting up of initial conditions, and then the program is started and runs. However, in an agent system (or, more generally a reactive system), the running system continues to interact with its environment, and so a test is not just the initial conditions, but also comprises the ongoing interactions of the system with its environment. One consequence of this is that conditions are controllable. If an agent system tests condition c at a certain point in time, and then tests that condition again later, then in general the environment might have changed c, and so we assume that all conditions can be controlled by the test environment. This means that, for instance, if we have a test (i.e. path) that involves two subsequent parts of the graph, G_1 and G_2, then the specific path taken through G_2 can be treated as being independently controllable from that taken through G_1.

We now seek to derive equations that calculate the smallest number of paths from S to E required such that all edges appear at least once in the set of paths.

In order to do this, it turns out that we need to also capture how many of these paths correspond to successful executions (go via Y) and how many go via N. Notation[8]: we define $\mathbf{p}(P)$ to be the number of paths required to cover all edges in the graph corresponding to program P. We also define $\mathbf{y}(P)$ (respectively $\mathbf{n}(P)$) to be the number of these paths that go via Y (respectively N). By construction we have that $\mathbf{p}(P) = \mathbf{y}(P) + \mathbf{n}(P)$.

Let us now consider each case in turn. The base case of a single action a is straightforward. In the graph above it corresponds to the sub-graph P being a single node a. To cover all edges in the graph we need two test cases: one path S-a-Y-E and one S-a-N-E. This reflects that an action a can either succeed or fail, and therefore requires two tests to cover these possibilities. Formally we have that $\mathbf{p}(a) = 2$, and that $\mathbf{y}(a) = \mathbf{n}(a) = 1$.

Next we consider $P_1; P_2$. Suppose that a sub-program P_1 requires $\mathbf{p}(P_1)$ tests (i.e. paths) to cover all edges, with $\mathbf{n}(P_1)$ of these tests leading to the failure of P_1, and the remaining $\mathbf{y}(P_1)$ tests leading to successful execution of P_1. Since P_1 is put in sequence with P_2, we have the control flow graph on the right.



We seek to derive an equation for $\mathbf{p}(P_1; P_2)$ (and for $\mathbf{y}(P_1; P_2)$ and $\mathbf{n}(P_1; P_2)$) in terms of the properties of P_1 and P_2. Let us firstly consider the case where $\mathbf{y}(P_1) \leq \mathbf{p}(P_2)$. In this case if we have enough tests to cover the edges of the sub-graph corresponding to P_2, then these tests are also sufficient to cover all edges of P_1 that result in a successful execution of P_1 (which lead to P_2). So to cover all edges of P_1 we need to add in enough tests to cover those executions that are failed, i.e. $\mathbf{n}(P_1)$. Therefore we have that:

$$\mathbf{p}(P_1; P_2) = \mathbf{n}(P_1) + \mathbf{p}(P_2) \tag{1}$$

$$\mathbf{y}(P_1; P_2) = \mathbf{y}(P_2) \tag{2}$$

$$\mathbf{n}(P_1; P_2) = \mathbf{n}(P_1) + \mathbf{n}(P_2) \tag{3}$$

We now consider the case where $\mathbf{y}(P_1) \geq \mathbf{p}(P_2)$. In this case if we have enough tests (i.e. paths) to cover the edges of the sub-graph corresponding to P_1, then these tests are also sufficient to cover all edges of P_2. We therefore have that $\mathbf{p}(P_1; P_2) = \mathbf{p}(P_1) = \mathbf{n}(P_1) + \mathbf{y}(P_1)$.

However, when considering $\mathbf{y}(P_1; P_2)$ and $\mathbf{n}(P_1; P_2)$ things become a little more complex. Since $\mathbf{y}(P_1) > \mathbf{p}(P_2)$, the edge from the sub-graph corresponding to P_1 that goes to the sub-graph corresponding to P_2 has more tests traversing it than are required to cover all edges of P_2. In effect, this leaves us with "excess" tests (paths), and we need to work out how many of these excess paths should be allocated to successful executions of P_2 (i.e. $\mathbf{y}(P_2)$), and how many to $\mathbf{n}(P_2)$.

[8] Colour is used to assist readability, but is not essential.

Consider the following example. Suppose that $P_1; P_2$ is such that[9] P_1 requires 5 tests to cover all edges (four successful, and hence available to test P_2, and one unsuccessful), and where P_2 only requires 2 tests to cover all edges. In this situation there are two additional tests that are required to test P_1 and which proceed to continue executing P_2. These two extra tests could correspond to failed executions of P_2, to successful executions of P_2, or to one successful and one failed execution. This means that, if we annotate each edge with the number of times that it is traversed by the set of tests[10], then the edge from Y_1 to the P_2 sub-graph is traversed 4 times, since the edge from P_1 to Y_1 traversed 4 times. The edge from P_2 to Y_2 could have either a 1, 2, or 3, and similarly the edge from P_2 to N could have either 3, 2, or 1 (see Figure 1 on the next page).

Returning to the analysis, in this case, where $\mathbf{y}(P_1) > \mathbf{p}(P_2)$, we define $\epsilon_1 + \epsilon_2 = \mathbf{y}(P_1) - \mathbf{p}(P_2)$. Then if we annotate each edge with the number of times that it is traversed by the tests, then the annotation on the edge from Y_1 to P_2 would be $\mathbf{p}(P_2) + \epsilon_1 + \epsilon_2$. If we now consider the edges *from* the sub-graph corresponding to P_2, then the edge to N (the number of executions where P_2 failed) would be annotated with $\mathbf{n}(P_2) + \epsilon_2$ and the edge to Y_2 would be annotated with $\mathbf{y}(P_2) + \epsilon_1$. This gives us the following equations:

$$\mathbf{p}(P_1; P_2) = \mathbf{n}(P_1) + \mathbf{y}(P_1) \tag{4}$$
$$\mathbf{y}(P_1; P_2) = \mathbf{y}(P_2) + \epsilon_1 \tag{5}$$
$$\mathbf{n}(P_1; P_2) = \mathbf{n}(P_1) + \mathbf{n}(P_2) + \epsilon_2 \tag{6}$$
$$\text{where} \quad \epsilon_1 + \epsilon_2 = \mathbf{y}(P_1) - \mathbf{p}(P_2)$$

Merging these cases with Equations 1, 2 and 3, we obtain the following. Derivation: for $\mathbf{y}()$ and $\mathbf{n}()$ observe that equations 2 and 3 are in the case where $\mathbf{y}(P_1) \leq \mathbf{p}(P_2)$ and hence $\epsilon_1 = \epsilon_2 = 0$, reducing the equations below to Equations 2 and 3, and if $\mathbf{y}(P_1) > \mathbf{p}(P_2)$ then the equations below are identical to Equations 5 and 6. For $\mathbf{p}(P_1; P_2)$ observe that if $\mathbf{y}(P_1) \leq \mathbf{p}(P_2)$ then the equation below reduces to Equation 1, and that if $\mathbf{y}(P_1) > \mathbf{p}(P_2)$ then the equation below reduces to Equation 4.

$$\mathbf{p}(P_1; P_2) = \mathbf{n}(P_1) + \max(\mathbf{y}(P_1), \mathbf{p}(P_2))$$
$$\mathbf{y}(P_1; P_2) = \mathbf{y}(P_2) + \epsilon_1$$
$$\mathbf{n}(P_1; P_2) = \mathbf{n}(P_1) + \mathbf{n}(P_2) + \epsilon_2$$
$$\text{where} \quad \epsilon_1 + \epsilon_2 = \max(0, \mathbf{y}(P_1) - \mathbf{p}(P_2))$$

Note that we don't have deterministic equations that compute $\mathbf{n}(P_1; P_2)$ and $\mathbf{y}(P_1; P_2)$. Instead, we have equations that permit a *range* of values, depending on how we choose to allocate the excess paths represented by $\epsilon_1 + \epsilon_2$ between the successful and unsuccessful executions of P_2.

[9] E.g. $P_1 = a_1 \triangleright a_2 \triangleright a_3 \triangleright a_4$ and $P_2 = a_5$.

[10] Note that for any internal node, the sum of annotations on incoming edges must equal the sum of annotations on outgoing edges, since all paths begin at S and terminate at E.

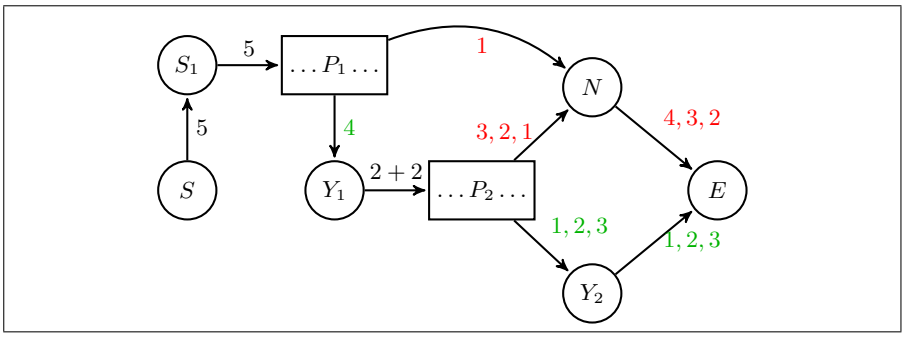
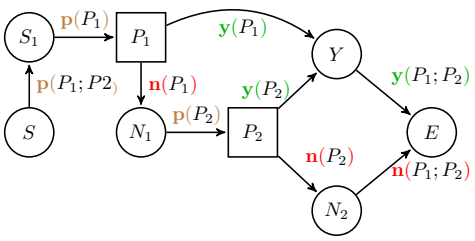


Fig. 1. Example for $P_1 = a_1 \triangleright a_2 \triangleright a_3 \triangleright a_4$ and $P_2 = a_5$.



Turning to $P_1 \triangleright P_2$ we perform a similar analysis. Note that the control glow graph for $P_1 \triangleright P_2$ has the same structure as that of $P_1; P_2$ except that N and Y are swapped (see Figure to the right). The simple case is when $\mathbf{n}(P_1) \leq \mathbf{p}(P_2)$, in which case the number of paths required to test (i.e. cover all edges of) P_2 also suffices to cover edges in P_1 when P_1 fails (for $P_1 \triangleright P_2$ it is when P_1 *fails* that P_2 is used). For this case we therefore have $\mathbf{p}(P_1 \triangleright P_2) = \mathbf{y}(P_1) + \mathbf{p}(P_2)$ and $\mathbf{y}(P_1 \triangleright P_2) = \mathbf{y}(P_1) + \mathbf{y}(P_2)$ and $\mathbf{n}(P_1 \triangleright P_2) = \mathbf{n}(P_2)$. Similar analysis for the more complex case gives the equations in Figure 2.

Finally, we consider goals. We begin with the simple case: a goal with a single *relevant* plan $g^{\{P_1\}}$. In this case either the goal immediately fails (due to the plan's context condition failing), or the plan is executed. If the plan is executed, then the goal succeeds exactly when the plan succeeds. Therefore we have: $\mathbf{n}(g^{\{P_1\}}) = 1 + \mathbf{n}(P_1)$, and $\mathbf{y}(g^{\{P_1\}}) = \mathbf{y}(P_1)$, and $\mathbf{p}(g^{\{P_1\}}) = 1 + \mathbf{p}(P_1)$. In the case where P_1 is *applicable*, then the context condition cannot fail, and we simply have $\mathbf{n}(g^{\{P_1\}}) = \mathbf{n}(P_1)$ and $\mathbf{p}(g^{\{P_1\}}) = \mathbf{p}(P_1)$.

For a goal with two *relevant* plans $g^{\{P_1,P_2\}}$ (henceforth abbreviated g^2), there are three non-overlapping possibilities: the plan fails immediately (neither context condition is true), or the first plan is selected, or the second plan is selected. If a plan is selected, then the plan is executed with the other plan as a (possible) backup option. Informally we can describe this as

$$g^2 = \text{fail or } P_1 \triangleright g^{P_2} \text{ or } P_2 \triangleright g^{P_1}$$

(where g^P is short hand for $g^{\{P\}}$). Which leads to the following equations.

$$\begin{aligned}
\mathbf{p}(g^2) &= 1 + \mathbf{p}(P_1 \triangleright g^{P_2}) + \mathbf{p}(P_2 \triangleright g^{P_1}) \\
\mathbf{y}(g^2) &= \mathbf{y}(P_1 \triangleright g^{P_2}) + \mathbf{y}(P_2 \triangleright g^{P_1}) \\
\mathbf{n}(g^2) &= 1 + \mathbf{n}(P_1 \triangleright g^{P_2}) + \mathbf{n}(P_2 \triangleright g^{P_1})
\end{aligned}$$

$$
\begin{aligned}
\mathbf{p}(a) &= 2 \qquad \mathbf{y}(a) = 1 \qquad \mathbf{n}(a) = 1 \\
\mathbf{p}(P_1; P_2) &= \mathbf{n}(P_1) + \max(\mathbf{y}(P_1), \mathbf{p}(P_2)) \\
\mathbf{y}(P_1; P_2) &= \mathbf{y}(P_2) + \epsilon_1 \\
\mathbf{n}(P_1; P_2) &= \mathbf{n}(P_1) + \mathbf{n}(P_2) + \epsilon_2 \\
&\quad \text{where } \epsilon_1 + \epsilon_2 = \max(0, \mathbf{y}(P_1) - \mathbf{p}(P_2)) \\
\mathbf{p}(P_1 \triangleright P_2) &= \mathbf{y}(P_1) + \max(\mathbf{n}(P_1), \mathbf{p}(P_2)) \\
\mathbf{y}(P_1 \triangleright P_2) &= \mathbf{y}(P_1) + \mathbf{y}(P_2) + \epsilon_3 \\
\mathbf{n}(P_1 \triangleright P_2) &= \mathbf{n}(P_2) + \epsilon_4 \\
&\quad \text{where } \epsilon_3 + \epsilon_4 = \max(0, \mathbf{n}(P_1) - \mathbf{p}(P_2)) \\
\mathbf{p}(g^{\{P\}}) &= \mathbf{1+}\,\mathbf{p}(P) \qquad \mathbf{y}(g^{\{P\}}) = \mathbf{y}(P) \qquad \mathbf{n}(g^{\{P\}}) = \mathbf{1+}\,\mathbf{n}(P) \\
\mathbf{p}(g^{\mathcal{P}}) &= \mathbf{1+} \sum_{P_i \in \mathcal{P}} \mathbf{y}(P_i) + \max(\mathbf{n}(P_i), \mathbf{p}(g^{\mathcal{P}\setminus\{P_i\}})) \\
\mathbf{y}(g^{\mathcal{P}}) &= \sum_{P_i \in \mathcal{P}} \mathbf{y}(P_i) + \mathbf{y}(g^{\mathcal{P}\setminus\{P_i\}}) + \epsilon_i \\
\mathbf{n}(g^{\mathcal{P}}) &= \mathbf{1+} \sum_{P_i \in \mathcal{P}} \mathbf{n}(g^{\mathcal{P}\setminus\{P_i\}}) + \epsilon_i' \\
&\quad \text{where } \epsilon_i + \epsilon_i' = \max(0, \mathbf{n}(P_i) - \mathbf{p}(g^{\mathcal{P}\setminus\{P_i\}})) \\
\mathbf{p}(\not{g}^{\mathcal{P}}) &= \mathbf{1+} \sum_{P \in \mathcal{P}} \mathbf{p}(P) \\
\mathbf{y}(\not{g}^{\mathcal{P}}) &= \sum_{P \in \mathcal{P}} \mathbf{y}(P) \\
\mathbf{n}(\not{g}^{\mathcal{P}}) &= \mathbf{1+} \sum_{P \in \mathcal{P}} \mathbf{n}(P)
\end{aligned}
$$

Fig. 2. Equations to calculate $\mathbf{p}(P)$, $\mathbf{y}(P)$ and $\mathbf{n}(P)$ when $\mathcal{P}$ is relevant plans. For applicable plans delete the grey shaded "$1+$".

In the case where we are dealing with *applicable* plans, the only difference is that the "1+" in the equations for $\mathbf{p}(g)$ and $\mathbf{n}(g)$ is deleted, since the plan cannot fail. This can be generalised for a goal with k plans (details omitted) resulting in the equations in Figure 2.

3.1 Removing Failure Handling

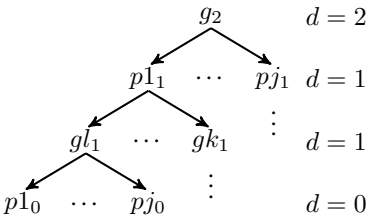
We now briefly consider what happens if we "turn off" failure handling, This is an interesting scenario to consider, because the all paths analysis of Winikoff & Cranefield [20] found that turning failure handling off reduced the number of tests required enormously. We use $\not{g}$ to denote a goal where failure handling is not used.

We firstly observe that without failure handling the equation for $\not{g}^{\{P\}}$ remains unchanged from $g^{\{P\}}$, since if the sole plan P fails, then there is no remaining plan available to recover.

However, for $\not{g}^{\{P_1,P_2\}}$ the equations are different. Instead of having (informally) $g^2 =$ fail or $P_1 \triangleright g^{P_2}$ or $P_2 \triangleright g^{P_1}$, we have simply $\not{g}^2 =$ fail or P_1 or P_2. Therefore the corresponding equations are simply: $\mathbf{p}(\not{g}^2) = 1 + \mathbf{p}(P_1) + \mathbf{p}(P_2)$, and $\mathbf{y}(\not{g}^2) = \mathbf{y}(P_1) + \mathbf{y}(P_2)$, and $\mathbf{n}(\not{g}^2) = 1 + \mathbf{n}(P_1) + \mathbf{n}(P_2)$. These generalise for $\not{g}^{\mathcal{P}}$ (where $\mathcal{P}$ denotes a set of plans), yielding the equations in Figure 2. As before, for $\mathcal{P}$ being the *applicable* plans, remove the "$1 +$" from the equations.

3.2 Simplifying for Uniform Programs

In order to compare with the all paths analysis of Winikoff & Cranefield [20] we consider *uniform* BDI programs, as they did. A uniform BDI program is one where all plan bodies have j sub-goals, all goals have k plans, and the tree is uniformly deep.



Applying these assumptions allows the equations to be simplified, since all sub-goals of a plan (respectively plans of a goal) have identical structure, and are hence interchangeable.

For example, in the equation for $\mathbf{p}(P_1; P_2)$, P_1 and P_2 are identical, so instead of $\mathbf{p}(P_1; P_2) = \mathbf{n}(P_1) + \max(\mathbf{y}(P_1), \mathbf{p}(P_2))$ we have $\mathbf{p}(P; P) = \mathbf{n}(P) + \max(\mathbf{y}(P), \mathbf{p}(P))$. Now, since $\mathbf{p}(P) > \mathbf{y}(P)$, we can replace $\max(\mathbf{y}(P), \mathbf{p}(P))$ with $\mathbf{p}(P)$. Therefore, we have that $\mathbf{p}(P; P) = \mathbf{n}(P) + \mathbf{p}(P)$. Since $\mathbf{p}(P) = \mathbf{y}(P) + \mathbf{n}(P)$ this is just $\mathbf{n}(P) + \mathbf{y}(P) + \mathbf{n}(P) = \mathbf{y}(P) + 2\mathbf{n}(P)$. This generalises to more than two sub-programs in sequence. Similar simplification can be applied to the other cases, yielding the equations shown in Figure 3 (ignore the last four equations for the moment).

However, uniform programs (as used by the all paths analysis [20]) actually have a mixture of actions and goals in plans, i.e. a plan (that is not a leaf) is of the form $P = a; g; a; g; a$ (for $k = 2$), not $g; g$. This means we need to derive equations for this form.

We begin by deriving $\mathbf{p}(a; g)$, $\mathbf{y}(a; g)$ and $\mathbf{n}(a; g)$, using the simplification that $\epsilon_1 = \epsilon_2 = 0$, since $\mathbf{y}(P_1) = \mathbf{y}(a) = 1$ and hence $\mathbf{p}(P_2) \geq 1$ so $\max(0, \mathbf{y}(P_1) - \mathbf{p}(P_2)) = 0$.

$$\begin{array}{lcll} \mathbf{p}(a; g) & = & \mathbf{n}(a) + \max(\mathbf{y}(a), \mathbf{p}(g)) & = 1 + \mathbf{p}(g) \quad \text{(since } \mathbf{p}(g) > \mathbf{y}(a) = 1\text{)} \\ \mathbf{y}(a; g) & = & \mathbf{y}(g) \text{ (since } \mathbf{p}(g) > \mathbf{y}(a) = 1\text{)} & \\ \mathbf{n}(a; g) & = & 1 + \mathbf{n}(g) & \end{array}$$

We then define $p^1 = a; g; a$ and derive $\mathbf{p}(p^1)$, $\mathbf{y}(p^1)$ and $\mathbf{n}(p^1)$. In deriving $\mathbf{y}(p^1)$ and $\mathbf{n}(p^1)$ we derive the upper and lower bounds (recall that the equations in Figure 2 specify a range, depending on how we split "excess" ($\mathbf{y}(P_1) - \mathbf{p}(P_2)$) between ϵ_1 and ϵ_2). We work out the upper bound for $\mathbf{y}(P_1)$ (respectively $\mathbf{n}(P_1)$) by assigning all the excess to ϵ_1 (respectively ϵ_2). We derive equations under the assumption that $\mathbf{y}(a; g) > 1$, and hence $\mathbf{y}(a; g) \geq \mathbf{p}(a) = 2$. This assumption holds when goals have more than one plan (i.e. $j > 1$), which is the case in Table 1.

$$\begin{array}{lcll} \mathbf{p}((a; g); a) & = & \mathbf{n}(a; g) + \max(\mathbf{y}(a; g), \mathbf{p}(a)) & \\ & = & \mathbf{n}(a; g) + \mathbf{y}(a; g) \quad = \mathbf{p}(a; g) & = 1 + \mathbf{p}(g) \\ \mathbf{y}((a; g); a) & \leq & \mathbf{y}(a) + \max(0, \mathbf{y}(a; g) - \mathbf{p}(a)) & \end{array}$$

$$
\begin{array}{lcl}
 & = & 1 + \mathbf{y}(a;g) - 2 \quad = \mathbf{y}(g) - 1 \\
\mathbf{y}((a;g);a) & \geq & \mathbf{y}(a) \;=\; 1 \\
\mathbf{n}((a;g);a) & \leq & \mathbf{n}(a;g) + \mathbf{n}(a) + \max(0, \mathbf{y}(a;g) - \mathbf{p}(a)) \\
 & = & (1 + \mathbf{n}(g)) + 1 + (\mathbf{y}(a;g) - 2) \quad = \mathbf{n}(g) + \mathbf{y}(g) \quad = \mathbf{p}(g) \\
\mathbf{n}((a;g);a) & \geq & \mathbf{n}(a;g) + \mathbf{n}(a) \quad = (1 + \mathbf{n}(g)) + 1 \quad = 2 + \mathbf{n}(g)
\end{array}
$$

We then note that $p^2 = a;g;a;g;a$ can be defined as $p^2 = (a;g);p^1$, and, more generally, $p^{k+1} = (a;g);p^k$.

$$
\begin{array}{lcl}
\mathbf{p}(p^{k+1}) & = & \mathbf{n}(a;g) + \max(\mathbf{y}(a;g), \mathbf{p}(p^k)) \\
 & = & \mathbf{n}(a;g) + \mathbf{p}(p^k) \quad \text{(since } \mathbf{p}(p^k) \geq \mathbf{y}(a;g)\text{)} \\
 & = & 1 + \mathbf{n}(g) + \mathbf{p}(p^k) \\
 & & \text{which can be generalised to} \\
 & = & k \times (1 + \mathbf{n}(g)) + 1 + \mathbf{p}(g) \\
\mathbf{y}(p^{k+1}) & \leq & \mathbf{y}(p^k) + \max(0, \mathbf{y}(a;g) - \mathbf{p}(p^k)) \\
 & = & \mathbf{y}(p^k) \quad \text{(since } \mathbf{p}(p^k) \geq \mathbf{y}(a;g)\text{)} \\
 & & \text{so eventually we just get } \mathbf{y}(p^1) \text{ which is } \ldots \\
 & = & \mathbf{y}(g) - 1 \\
\mathbf{y}(p^{k+1}) & \geq & \mathbf{y}(p^k) \quad \geq \mathbf{y}(p^{k-1}) \quad \geq 1 \\
\mathbf{n}(p^{k+1}) & = & \mathbf{n}(a;g) + \mathbf{n}(p^k) + \max(0, \mathbf{y}(a;g) - \mathbf{p}(p^k)) \\
 & = & (1 + \mathbf{n}(g)) + \mathbf{n}(p^k) \text{ (since } \mathbf{p}(p^k) \geq \mathbf{y}(a;g)\text{)} \\
 & = & k \times (1 + \mathbf{n}(g)) + \mathbf{n}(p^1)
\end{array}
$$

The yields the last four equations of Figure 3, which are required to calculate the testability of uniform BDI programs. Note that in the last equation, since $\mathbf{n}(p^1) \geq 2 + \mathbf{n}(g)$ and $\mathbf{n}(p^1) \leq \mathbf{p}(g)$, we also have a range for $\mathbf{n}(p^k)$.

4 All-edges vs. All-paths

In the previous section we derived equations that tell us how many tests (paths) are required to ensure adequate coverage of a BDI program with respect to the all *edges* criterion. We now use these equations to compare the all edges criterion against the all paths criterion. We know that the all paths criterion requires more tests to be satisfied, but how many more? Since comparing (complex) formulae is not easy, we follow the approach of Winikoff & Cranefield, and instantiate the formulae with a number of plausible values, to obtain actual numbers that can be compared. We use the same scenarios (i.e. parameters) that they used.

In order to derive the All Edges numbers in Table 1 the equations of Figure 2 were implemented as a Prolog program that computed (non-deterministically) the values of $\mathbf{p}(P)$, $\mathbf{y}(P)$ and $\mathbf{n}(P)$ for any given BDI program P. Additionally, code was written to generate a uniform BDI program P, given values for j, k, and d. This was used to generate the full uniform program P for the first three cases in Table 1, and then compute $\mathbf{p}(P)$ for the generated BDI program. The last case exhausted Prolog's stack.

Additionally, the equations of Figure 3 were implemented as a Scheme program that computed $\mathbf{p}()$, $\mathbf{y}()$, and $\mathbf{n}()$ for given values of j, k, and d. These were used to calculate

$$
\begin{aligned}
\mathbf{p}(P_1;\ldots;P_j) &= \mathbf{y}(P) + j \times \mathbf{n}(P)\\
\mathbf{y}(P_1;\ldots;P_j) &= \mathbf{y}(P)\\
\mathbf{n}(P_1;\ldots;P_j) &= j \times \mathbf{n}(P)\\
\mathbf{p}(P_1 \triangleright \ldots \triangleright P_k) &= \mathbf{n}(P) + k \times \mathbf{y}(P)\\
\mathbf{y}(P_1 \triangleright \ldots \triangleright P_k) &= k \times \mathbf{y}(P)\\
\mathbf{n}(P_1 \triangleright \ldots \triangleright P_k) &= \mathbf{n}(P)\\
\mathbf{p}(g^{\{P\}}) &= 1 + \mathbf{p}(P)\\
\mathbf{y}(g^{\{P\}}) &= \mathbf{y}(P)\\
\mathbf{n}(g^{\{P\}}) &= 1 + \mathbf{n}(P)\\
\mathbf{p}(g^{\mathcal{P}}) &= 1 + |\mathcal{P}| \times (\mathbf{y}(P) + \mathbf{p}(g^{\mathcal{P}\setminus\{P_i\}}))\\
\mathbf{y}(g^{\mathcal{P}}) &= |\mathcal{P}| \times (\mathbf{y}(P) + \mathbf{y}(g^{\mathcal{P}\setminus\{P_i\}}))\\
\mathbf{n}(g^{\mathcal{P}}) &= 1 + |\mathcal{P}| \times \mathbf{n}(g^{\mathcal{P}\setminus\{P_i\}})\\
\mathbf{p}(\not{g}^{\mathcal{P}}) &= 1 + |\mathcal{P}| \times \mathbf{p}(P)\\
\mathbf{y}(\not{g}^{\mathcal{P}}) &= |\mathcal{P}| \times \mathbf{y}(P)\\
\mathbf{n}(\not{g}^{\mathcal{P}}) &= 1 + |\mathcal{P}| \times \mathbf{n}(P)\\
\mathbf{p}(p^{k+1}) &= k \times (1 + \mathbf{n}(g)) + 1 + \mathbf{p}(g)\\
\mathbf{y}(p^{k+1}) &\leq \mathbf{y}(g) - 1\\
\mathbf{y}(p^{k+1}) &\geq 1\\
\mathbf{n}(p^{k+1}) &= k \times (1 + \mathbf{n}(g)) + \mathbf{n}(p^1)
\end{aligned}
$$

Fig. 3. Equations to calculate $\mathbf{p}(P)$, $\mathbf{y}(P)$ and $\mathbf{n}(P)$, simplified for uniform programs, where p^{k+1} denotes a program of the form $a; g; a; \ldots a; g; a$ with $k+1$ goals ($k \geq 0$).

values of $\mathbf{p}()$. These values matches those computed by the Prolog program for the first three cases, and provided the values for the fourth case ($d = 3$, $j = 3$, $k = 4$ for which Prolog ran out of stack space).

Table 1 contains the results for these illustrative comparison cases (ignore the right-most column for now). The left part of the Table (Parameters, Number of goals, plans, and actions, and All Paths) are taken from the all paths analysis of Winikoff & Cranefield [20]. The right part (All Edges) is the new numbers from this work.

Comparing the results we make a number of observations. Firstly, as expected, the number of tests required to adequately test a given BDI program P with respect to the all edges test adequacy criterion is lower than the number of tests required with respect to the all paths criterion. However, what is interesting is that the numbers are very much lower (e.g. a few thousand compared with more than 2×10^{107}). Specifically, the number of tests required with respect to the all edges criterion is sufficiently small to be feasible. For instance, in the third case ($j = 2$, $k = 3$, $d = 4$) where the (uniform) BDI program has 259 goals and 518 plans, corresponding to a non-trivial agent program, the number of required test cases is less than 1600.

However, it is worth emphasising that the all edges criterion, even for traditional software, is regarded as a *minimum*. Additionally, it can be argued that agents, which

Params			Number of ...			All Paths		All Edges				All Edges
								$\mathbf{p}(g)$		$\mathbf{p}(\not g)$		
j	k	d	goals	plans	actions	$n^{\checkmark}(g)$	$n^{\times}(g)$	relev.	applic.	relev.	applic.	$\mathbf{q}(Q)$
2	2	3	21	42	62 (13)	6.33×10^{12}	1.82×10^{13}	141	78	85	64	62
3	3	3	91	273	363 (25)	1.02×10^{107}	2.56×10^{107}	6,391	2,961	469	378	363
2	3	4	259	518	776 (79)	1.82×10^{157}	7.23×10^{157}	1,585	808	1,037	778	776
3	4	3	157	471	627 (41)	3.13×10^{184}	7.82×10^{184}	10,777	4,767	799	642	627

Table 1. Comparison of All Paths and All Edges analyses. The first number under "actions" (e.g. 62) is the number of actions in the tree, the second (e.g. 13) is the number of actions in a single execution where no failures occur. For All Edges there are four numbers: the first two are the (normal) case where failure handling is used to re-post a goal in the event that a plan fails. The next two are the case where failure handling is disabled, so if a plan fails, the parent goal fails as well. The columns labelled "relev." and "applic." are where the plans associated with a goal are respectively the *relevant* plans (so a goal can fail even though there are still untried plans), and the *applicable* plans.

are situated in an environment that is typically non-episodic, might be more likely than traditional software to be affected by the history of their interaction with the environment [20, Section 1.1], which means that the all paths criterion is more relevant (since a path includes history, and requiring all paths insists that different histories are covered when testing).

We now turn to consider the four cases under All Edges, i.e. the effects of disabling failure handling, and allowing goals to fail even when there are remaining plans. Whereas a key finding of Winikoff & Cranefield was that failure handling made an enormous difference, in our analysis we found the opposite. This does not reflect a disagreement with their analysis, but a difference in the characteristics of all paths vs. all edges. Adding failure handling has the effect of extending paths that would otherwise fail. This means that enabling failure handling increases the number of paths. However, for the all edges criterion, we do not need to cover all paths, only all edges, so the additional paths created by enabling failure handling do not require a commensurate increase in the number of tests required to cover all edges.

Finally, we consider the difference between the set of plans associated with a goal being the *relevant* and being the *applicable* plan set. Interestingly, this makes a difference, and surprisingly, in some cases it makes more of a difference than enabling failure handling! For example, in the third example case ($j = 2$, $k = 3$, $d = 4$) where *more* tests are required without failure handling (1037) than with failure handling, but where the plans are the applicable plan set (808). Note that the all paths analysis considered the j plans associated with each goal to be *applicable*.

5 BDI vs. Procedural

The previous section considered the question of whether testing BDI agent programs was *hard*. We now consider the question of whether it is *harder*, i.e. we compare the number of tests required to adequately test a BDI agent program (with respect to the all

edges criterion) with the number of tests required to adequately test an equivalent-sized (abstract) procedural program.

We choose to compare equivalently-sized programs for the simple reason that, in general, a larger program (procedural or BDI) will require more tests. So in order to compare procedural and BDI programs we need to keep the size fixed. The particular measure of size that we use is the number of primitive elements, actions for BDI programs, primitive statements for procedural programs.

Following Winikoff & Cranefield [20] we define an abstract procedural program as (we use Q to avoid confusion with BDI programs P):

$$Q ::= s \mid Q + Q \mid Q; Q$$

In other words, the base case is a statement s, and a compound program can be a combination of programs either in sequence ($Q_1; Q_2$), or as an alternative choice ($Q_1 + Q_2$). Note that for our analysis we do not need to model the condition on the choice, so the program "**if** c **then** Q_1 **else** Q_2" is simply represented as a choice between Q_1 and Q_2, i.e. $Q_1 + Q_2$. Note that loops are modelled as a choice between looping and not looping (following standard practice [13, p.408] we only consider loops to be executed once, or zero times). Mapping these programs to control-flow graphs is straightforward, and a program is mapped to a single-entry and single-exit graph. Note that these procedural programs do not include failure handling: we are comparing the

We now consider how many tests (i.e. paths) are required to cover all edges in the graph corresponding to a procedural program Q. We denote this number (i.e. the testability of program Q with respect to the all edges criterion) by $\mathbf{q}(Q)$. There are three cases. In the base case, a single statement, a single path suffices to cover both edges. In the case of an alternative, each path either traverses the sub-graph corresponding to Q_1, or the sub-graph corresponding to Q_2. Therefore the number of paths required to cover all edges in the graph corresponding to $Q_1 + Q_2$ is the *sum* of the number of paths required for each of the two sub-graphs, i.e. $\mathbf{q}(Q_1 + Q_2) = \mathbf{q}(Q_1) + \mathbf{q}(Q_2)$. Turning to a sequence $Q_1; Q_2$, suppose that we require $\mathbf{q}(Q_1)$ tests to cover all edges in Q_1, and, respectively, $\mathbf{q}(Q_2)$ paths to cover all edges in Q_2. Note that each path traverses the sub-graph corresponding to Q_1, and then continues to traverse the sub-graph corresponding to Q_2. This means that each path "counts" towards both Q_1 and Q_2, so the smallest number of paths that might be able to cover all edges is just the maximum of the number of paths required to test each of the two sub-graphs ($\mathbf{q}(Q_1; Q_2) = \max(\mathbf{q}(Q_1), \mathbf{q}(Q_2))$).

However, this assumes that paths used to cover the part of the control-flow graph corresponding to Q_1 can be "reused" effectively to cover the Q_2 part of the graph. This may not be the case, and since conditions are not controllable (the environment cannot change conditions while the program is running), we cannot make this assumption. So although it might be possible that only $\max(\mathbf{q}(Q_1), \mathbf{q}(Q_2))$ tests (i.e. paths) would suffice to cover all edges in the control flow graph corresponding to $Q_1; Q_2$, it may also be the case that more tests are required. In the worse case it might be that the set of tests designed to cover all edges of Q_1 all take the same path through Q_2, in which case we would require an additional $\mathbf{q}(Q_2) - 1$ tests to cover the rest of the sub-graph

corresponding to Q_2. This yields the following definition:

$$\begin{aligned} \mathbf{q}(s) &= 1 \\ \mathbf{q}(Q_1; Q_2) &\geq \max(\mathbf{q}(Q_1), \mathbf{q}(Q_2)) \\ \mathbf{q}(Q_1; Q_2) &\leq \mathbf{q}(Q_1) + \mathbf{q}(Q_2) - 1 \\ \mathbf{q}(Q_1 + Q_2) &= \mathbf{q}(Q_1) + \mathbf{q}(Q_2) \end{aligned}$$

We define the *size* of a program Q (denoted by $|Q|$) as being the number of statements. It can then be shown that for a procedural program Q of size m it is the case that $\mathbf{q}(Q) \leq m$.

Lemma 1. $\mathbf{q}(Q) \leq |Q|$.
Proof by induction: *Base case: size 1, so* $Q = s$, *and* $\mathbf{q}(s) = 1 \leq 1$. *Induction: suppose* $\mathbf{q}(Q) \leq |Q|$ *for* $|Q| < m$, *need to show it also holds for* $|Q| = m$. *Observe that* $\mathbf{q}(Q_1; Q_2) < \mathbf{q}(Q_1 + Q_2)$, *so we only need to show that* $\mathbf{q}(Q_1 + Q_2) \leq |Q_1| + |Q_2|$, *and the case for* $\mathbf{q}(Q_1; Q_2)$ *then follows. So, consider the case where* $Q = Q_1 + Q_2$, *hence* $\mathbf{q}(Q) = \mathbf{q}(Q_1) + \mathbf{q}(Q_2)$. *By the induction hypothesis we have that* $\mathbf{q}(Q_1) \leq |Q_1|$ *and* $\mathbf{q}(Q_2) \leq |Q_2|$ *and so* $\mathbf{q}(Q_1 + Q_2) = \mathbf{q}(Q_1) + \mathbf{q}(Q_2) \leq |Q_1| + |Q_2| = |Q|$. □

In other words, the number of paths (tests) required to cover all edges is at most the number of statements in the program. By contrast, to cover all paths, the number of tests required is approximately $3^{m/3}$ [20, page 109].

The rightmost column of Table 1 shows the number of tests (paths) required to test a procedural program Q of the same size as the BDI program in question for that row. Following Winikoff & Cranefield, we define size in terms of the number of actions (BDI) and statements (procedural), so, for example, the first row of Table 1 concerns a BDI goal-plan tree containing 62 actions (with $j = k = 2$ and $d = 3$), and a procedural program containing 62 statements.

We observe that the case with no failure handling and where $\mathcal{P}$ is *applicable* plans (i.e. the rightmost of the four numbers) is very close to $\mathbf{q}(Q)$. On the other hand, enabling failure handling does, for some cases, result in significantly more tests being required to adequately test the program. For example, 6,391 vs. 363, or 10,777 vs. 627. Both these cases have $j = 3$, whereas for the other two cases where $j = 2$ the difference is smaller. So we conclude that, especially where failure handling exists (which is the case for most BDI agent programming languages), and where goals have multiple plans available, then testing a BDI agent program is indeed harder than testing an equivalently-sized procedural program.

6 Conclusion

We considered the question of whether testing of BDI agent programs is feasible by quantifying the number of tests required to adequately test a given BDI agent program with respect to the all edges criterion. Our findings extend the earlier analysis of this question with respect to the all paths criterion to give a more nuanced understanding of the difficulty of testing BDI agents.

One key conclusion is that the number of tests required to satisfy the all edges criterion is not just lower (as expected) but very much lower (e.g. $> 2 \times 10^{107}$ vs. around $6,400$). Indeed, the number of tests required is sufficiently small to be feasible, although we do need to emphasise that all edges is generally considered to be a *minimal* requirement, and that there are arguments for why it is less appropriate for agent systems.

We also found that the introduction of failure handling did not make as large a difference for the all edges criterion, as it did for the all paths analysis.

When comparing BDI programs to procedural programs, our conclusion lends strength to the earlier result of Winikoff & Cranefield. They found that BDI agent programs were *harder* to test than equivalently sized procedural programs (with respect to the all paths criterion). We found that this is also the case for the all edges criterion, but only where goals had more than two plans.

Our overall conclusion is that BDI programs do indeed seem to be *harder* to test than procedural programs of equivalent size. However, whether it is feasible to test (whole) BDI programs remains unsettled. The all paths analysis (which is known to be pessimistic) concluded that BDI programs could not be feasibly tested. On the other hand, the all edges analysis (known to be optimistic) concluded that BDI programs could be feasibly tested. Further work is required.

Other future work includes applying these calculations to real programs, and continuing the development of formal methods for assuring the behaviour of agent-based systems [23, 3, 16, 6, 8, 10, 7].

References

1. R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, 2005.
2. R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Tools and Applications*. Springer, 2009.
3. R. H. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge. Model checking AgentSpeak. In *Autonomous Agents and Multiagent Systems (AAMAS)*, pages 409–416, 2003.
4. M. E. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.
5. M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4:349–355, 1988.
6. M. Dastani, K. V. Hindriks, and J.-J. C. Meyer, editors. *Specification and Verification of Multi-agent systems*. Springer, Berlin/Heidelberg, 2010.
7. L. A. Dennis, M. Fisher, N. K. Lincoln, A. Lisitsa, and S. M. Veres. Practical verification of decision-making in agent-based autonomous systems. *Automated Software Engineering*, 2014. 55 pages.
8. L. A. Dennis, M. Fisher, M. P. Webster, and R. H. Bordini. Model checking agent programming languages. *Automated Software Engineering Journal*, 19(1):3–63, March 2012.
9. E. E. Ekinci, A. M. Tiryaki, Ö. Çetin, and O. Dikenelli. Goal-oriented agent testing revisited. In M. Luck and J. J. Gomez-Sanz, editors, *Agent-Oriented Software Engineering IX*, volume 5386 of *Lecture Notes in Computer Science*, pages 173–186, Berlin/Heidelberg, 2009. Springer.
10. M. Fisher, L. Dennis, and M. Webster. Verifying autonomous systems. *Communications of the ACM*, 56(9):84–93, 2013.

11. J. J. Gomez-Sanz, J. Botía, E. Serrano, and J. Pavón. Testing and debugging of MAS interactions with INGENIAS. In M. Luck and J. J. Gomez-Sanz, editors, *Agent-Oriented Software Engineering IX*, volume 5386 of *Lecture Notes in Computer Science*, pages 199–212, Berlin/Heidelberg, 2009. Springer.
12. P. Jorgensen. *Software Testing: A Craftsman's Approach*. CRC Press, second edition edition, 2002.
13. A. P. Mathur. *Foundations of Software Testing*. Pearson, 2008. ISBN 978-81-317-1660-1.
14. C. D. Nguyen, A. Perini, and P. Tonella. Experimental evaluation of ontology-based test generation for multi-agent systems. In M. Luck and J. J. Gomez-Sanz, editors, *Agent-Oriented Software Engineering IX*, volume 5386 of *Lecture Notes in Computer Science*, pages 187–198, Berlin/Heidelberg, 2009. Springer.
15. L. Padgham, Z. Zhang, J. Thangarajah, and T. Miller. Model-based test oracle generation for automated unit testing of agent systems. *IEEE Transactions on Software Engineering*, 39:1230–1244, 2013.
16. F. Raimondi and A. Lomuscio. Automatic verification of multi-agent systems by model checking via ordered binary decision diagrams. *J. Applied Logic*, 5(2):235–251, 2007.
17. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. V. de Velde and J. Perrame, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96)*, pages 42–55. Springer Verlag, 1996. LNAI, Volume 1038.
18. A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *Principles of Knowledge Representation and Reasoning, Proceedings of the Second International Conference*, pages 473–484. Morgan Kaufmann, 1991.
19. R. Vieira, Á. Moreira, M. Wooldridge, and R. H. Bordini. On the formal semantics of speech-act based communication in an agent-oriented programming language. *Journal of Artificial Intelligence Research (JAIR)*, 29:221–267, 2007.
20. M. Winikoff and S. Cranefield. On the testability of BDI agent systems. *Journal of Artificial Intelligence Research (JAIR)*, 51:71–131, 2014.
21. M. Winikoff and S. Cranefield. On the testability of BDI agent systems (extended abstract). In *Journal track of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4217–4221, 2015.
22. M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative & procedural goals in intelligent agent systems. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 470–481, Toulouse, France, 2002. Morgan Kaufmann.
23. M. Wooldridge, M. Fisher, M.-P. Huget, and S. Parsons. Model checking multi-agent systems with MABLE. In *Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 952–959, 2002.
24. Z. Zhang, J. Thangarajah, and L. Padgham. Automated unit testing for agent systems. In *Second International Working Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, pages 10–18, 2007.
25. H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, Dec. 1997.

Reasoning about the Executability of Goal-Plan Trees

Yuan Yao[1], Lavindra de Silva[2], and Brian Logan[3]

[1] School of Computer Science
University of Nottingham
yvy@cs.nott.ac.uk
[2] Institute for Advanced Manufacturing
Faculty of Engineering
University of Nottingham
lavindra.desilva@nottingham.ac.uk
[3] School of Computer Science
University of Nottingham
bsl@cs.nott.ac.uk

Abstract. User supplied domain control knowledge in the form of hierarchically structured agent plans is at the heart of a number of approaches to reasoning about action. This knowledge encodes the "standard operating procedures" of an agent for responding to environmental changes, thereby enabling fast and effective action selection. This paper develops mechanisms for reasoning about a set of hierarchical plans and goals, by deriving "summary information" from the conditions on the execution of the basic actions forming the "leaves" of the hierarchy. We provide definitions of (necessary and contingent) pre-, in-, and postconditions of goals and plans that are consistent with the conditions of the actions forming a plan. Our definitions extend previous work with an account of both deterministic and non-deterministic actions, and with support for specifying that actions and goals within a (single) plan can execute concurrently. Based on our new definitions, we also specify requirements that are useful in scheduling the execution of steps in a set of goal-plan trees. These requirements essentially define conditions that must be protected by any scheduler that interleaves the execution of steps from different goal-plan trees.

1 Introduction

User supplied domain control knowledge in the form of hierarchically structured agent plans is at the heart of a number of approaches to reasoning about action. This knowledge encodes the "standard operating procedures" of an agent for responding to environmental changes, thereby enabling fast and effective action selection. Various lines of previous work have exploited such control knowledge, including multi-agent coordination [7, 8], interleaved plan execution in single-agent systems [16, 15], heuristic approaches to speeding up classical planning [3,

11, 5], and approaches to synthesising desirable primitive and abstract plans [12, 9].

This paper develops mechanisms for reasoning about a set of hierarchical plans and goals, by deriving "summary information" from the conditions on the execution of the basic actions forming the "leaves" of the hierarchy. We provide definitions of necessary and contingent pre-, in-, and postconditions of goals and plans that are consistent with the conditions of the (possibly nondeterministic) actions forming a plan. Such information is useful when writing agent programs, e.g. when deciding which goal-plan tree is the minimally interfering "building block" to include within a new plan in order to bring about a desired postcondition. In addition to summarising the "static" properties of a single goal-plan tree, we also define requirements that are useful in scheduling the execution of steps in a set of goal-plan trees. While goal-plan trees are are most commonly used to represent a BDI agent's domain knowledge, the mechanisms we present could equally be used to represent and reason about the executability of HTN (Hierarchical Task Network) planning [10] structures, e.g., to synthesise new HTN recipes from existing task networks. HTN and BDI systems are closely related in terms of syntax and semantics, making it possible to translate between the two representations [13].

The paper extends the most closely related strands of work in the literature, i.e., [7, 8, 16, 15] in two main ways. Like us, these authors also derive summary information from a set of hierarchical plans, and use that information to find a schedule for the concurrent execution of a given set of top-level goals. Our first extension is an account of both deterministic and non-deterministic primitive actions, and the second is the ability to specify that actions and goals within a (single) plan can execute concurrently. We also contribute novel corresponding definitions for the conditions that must be protected by any scheduler that interleaves the execution of steps from different goal-plan trees.

The remainder of this paper is organised as follows. In Section 2 we discuss closely related work from the literature. In Section 3, we define the static', necessary and contingent conditions of actions, plans and goals. Then, in Section 4 we define the corresponding dynamic' notions, which specify the conditions that must be taken into account when scheduling. Finally, in Section 5, we conclude and identify directions for future work.

2 Related Work

Our approach is closely related to two previous strands of work in the literature. The first is that of Clement et al. [7, 8], where algorithms are presented for deriving "summary" information from user-supplied hierarchical plans belonging to the various agents in a multi-agent system. The derived knowledge is then used to find a schedule that can coordinate the agents at runtime. The work of Thangarajah et al. [16, 15, 14] is similar, though they focus on the single-agent case. They describe an approach based on summary information that coordinates the various goal-plan trees of a single agent, in order to exploit positive

interactions between them and to avoid negative interactions; both of these involve reasoning about necessary and possible summary conditions for different ways of achieving a goal. They give algorithms for scheduling goal-plan trees, e.g., to determine whether a newly adopted (sub)goal will definitely be safe to execute without conflicts, or will definitely result in conflicts. In the latter case, Thangarajah et al. suspend the goal until it is safe to execute it.

An important difference between the work of Thangarajah et al. and that of Clement et al. is that the former defines the necessary post-condition of a goal or plan as the effects that are necessarily brought about at any (even an intermediate) stage during the goal's or plan's possible executions, whereas a necessary post-condition in the latter work represents only those effects that necessarily hold at the end of all executions. We incorporate both these notions in our approach; the former notion corresponds to our requirements that must hold for the successful execution of a set of goal-plan trees.

Another important difference involves a special case in which a plan has a step that makes a later step's associated descendant' (sub)plan inapplicable. To address this, [8] assumes that any such conflict might be resolvable later, during the scheduling phase, by inserting an available (concurrent) plan possibly one belonging to a different agent that asserts a suitable post-condition.[4] On the other hand, we disallow such conflicts in order to define a "local" notion of a contingent condition, which does not rely on other concurrent plans.

Our work is also related to that of de Silva et al. [9], who focus on how summary information could be used for the synthesis of desirable "abstract plans". To this end, the authors briefly describe how the above strands of work could be extended to support the specification of variables in agent programs, i.e., to a restricted first-order language. While de Silva et al. also support basic actions, they are deterministic, and cannot be overlapped with other actions.

3 Goal-Plan Trees

As in [16, 14] we use *goal-plan trees* to represent the relations between goals, plans and actions, and to reason about the interactions between intentions. The root of a goal-plan tree is a top-level goal[5] (goal node), and its children are the plans that can be used to achieve the goal (plan nodes). Plans may in turn contain subgoals (goal nodes), giving rise to a tree structure representing all possible ways an agent can achieve the top-level goal.

In [16, 14] goal-plan trees contain only goals and plans. We extend their definition of goal-plan trees to allow primitive actions in plans in addition to subgoals as in [2]. Plans thus consist of a sequence of steps which may contain actions. Figure 1 shows the BNF syntax of extended goal-plan trees. A *GoalType* is a template for a goal. A *GoalInstance* is created when an agent chooses to

[4] This assumption is related to the Modal Truth Criterion [6].

[5] We assume a procedural interpretation of goals ('goals to do' rather than goals to achieve a state). It is straightforward to adapt the definitions below for declarative goals.

pursue a particular instance of goal-type. Similarly, a *PlanType* is a template for a plan, and a *PlanInstance* is created when the agent executes a particular plan. In addition, we introduce an *ActionType* as a template for an action, and an *ActionInstance* is created when a particular action is chosen for execution by the agent. *GoalTypeName*, *PlanTypeName* and *ActionTypeName* are labels that indicate the type of the goal, the plan or the action respectively. *Plans* represents the set of plan-types that may be used to satisfy a goal of the corresponding *GoalType*.

$$
\begin{aligned}
\langle \text{GoalType} \rangle &::= \langle \text{GoalTypeName} \rangle\ \langle \text{Precondition} \rangle\ \langle \text{In-condition} \rangle\ \langle \text{Postcondition} \rangle \\
&\quad\ \langle \text{Plans} \rangle \\
\langle \text{GoalTypeName} \rangle &::= \langle \text{Label} \rangle \\
\langle \text{Plans} \rangle &::= \langle \text{PlanTypeName} \rangle\ (,\ \langle \text{PlanTypeName} \rangle)^* \\
\\
\langle \text{PlanType} \rangle &::= \langle \text{PlanTypeName} \rangle\ \langle \text{Precondition} \rangle\ \langle \text{In-condition} \rangle\ \langle \text{Postcondition} \rangle \\
&\quad\ \langle \text{PlanBody} \rangle \\
\langle \text{PlanTypeName} \rangle &::= \langle \text{Label} \rangle \\
\langle \text{PlanBody} \rangle &::= \langle \text{ExecutionStep} \rangle\ (;\ \langle \text{ExecutionStep} \rangle)^* \\
\langle \text{ExecutionStep} \rangle &::= \langle \text{ActionTypeName} \rangle \mid \langle \text{GoalTypeName} \rangle \\
&\quad\ \mid (\langle \text{ExecutionStep} \rangle \parallel \langle \text{ExecutionStep} \rangle) \\
\\
\langle \text{ActionType} \rangle &::= \langle \text{ActionTypeName} \rangle\ \langle \text{Precondition} \rangle\ \langle \text{In-condition} \rangle\ \langle \text{Postcondition} \rangle \\
\langle \text{ActionTypeName} \rangle &::= \langle \text{Label} \rangle \\
\\
\langle \text{Precondition} \rangle &::= \epsilon \mid \langle \text{Condition} \rangle\ (,\ \langle \text{Condition} \rangle)^* \\
\langle \text{In-condition} \rangle &::= \epsilon \mid \langle \text{Condition} \rangle\ (,\ \langle \text{Condition} \rangle)^* \\
\langle \text{Postcondition} \rangle &::= \epsilon \mid \langle \text{Condition} \rangle\ (,\ \langle \text{Condition} \rangle)^* \\
\langle \text{Condition} \rangle &::= \langle \text{Statement} \rangle \mid \mathit{NOT}\ \langle \text{Statement} \rangle \\
\langle \text{Statement} \rangle &::= \mathit{string} \mid \langle \text{Variable} \rangle = \langle \text{Value} \rangle \\
\langle \text{Label} \rangle &::= \mathit{unique\ string} \\
\langle \text{Variable} \rangle &::= \mathit{unique\ string} \\
\langle \text{Value} \rangle &::= \mathit{string} \\
\\
\langle \text{GoalInstance} \rangle &::= \langle \text{InstanceName} \rangle\ \langle \text{GoalType} \rangle \\
\langle \text{PlanInstance} \rangle &::= \langle \text{InstanceName} \rangle\ \langle \text{PlanType} \rangle \\
\langle \text{ActionInstance} \rangle &::= \langle \text{InstanceName} \rangle\ \langle \text{ActionType} \rangle \\
\langle \text{InstanceName} \rangle &::= \langle \text{Label} \rangle
\end{aligned}
$$

Fig. 1. BNF Syntax of goal-plan trees with actions

Goals, plans and actions have pre-, in-, and postconditions. Pre- and postconditions specify respectively the states of the environment which must hold immediately before the action, plan, or goal is executed, and which are brought about by executing the action, plan, or goal. In-conditions specify the states of the environment which must hold for the duration of the execution of the action, plan, or goal. In-conditions of plans and goals are thus relevant when their

associated actions are interleaved or overlapped, and in-conditions of actions are relevant when they are overlapped.

We model the environment using a set of propositions Φ, and define pre-, in- and postconditions of a goal-plan tree node η (an action, plan or goal) as sets of literals (elements of $\Phi^+ = \Phi \cup \{\neg p \mid p \in \Phi\}$) as follows.

Precondition: a precondition is a set of literals $\phi = pre(\eta)$, $\phi \subseteq \Phi^+$ that must be true for η to begin execution (where η is an action or plan), or for η to be achieved (where η is a goal).

In-condition: an in-condition is a set of literals $\upsilon = in(\eta)$, $\upsilon \subseteq \Phi^+$ that must hold during the execution of η (where η is an action or plan), or during the pursuit of η (where η is a goal); if any of the literals in υ becomes false during execution, the action, plan or goal is aborted with failure.

Postcondition: a postcondition (or effect) is a set of literals $\psi = post(\eta)$, $\psi \subseteq \Phi^+$ that are or may be made true by executing η (where η is an action or plan), or by achieving η (where η is a goal).

We distinguish two types of pre-, in- and postconditions: necessary and contingent. A *necessary* (or universal) condition must hold for all executions of an action or plan or for all ways of achieving a goal, while a *contingent* (or existential) condition must hold for some executions of the action or plan or some ways of achieving the goal. We denote the necessary and contingent preconditions as $pre_n(\eta)$ and $pre_c(\eta)$, where η is an action, plan or goal, and stipulate that $pre(\eta) = pre_n(\eta) \cup pre_c(\eta)$. Similarly, we denote necessary and contingent in-conditions as $in_n(\eta)$ and $in_c(\eta)$, and necessary and contingent postconditions as $post_n(\eta)$ and $post_c(\eta)$, and stipulate that $in(\eta) = in_n(\eta) \cup in_c(\eta)$ and $post(\eta) = post_n(\eta) \cup post_c(\eta)$. The necessary and contingent postconditions of an action, plan, or goal are always disjoint, and the same applies to necessary and contingent in-conditions, and to necessary and contingent preconditions.

While the relevant pre-, in- and postconditions form part of the definition of an action, the conditions of plans and goals are derived from the conditions of their actions and subgoals (in the case of plans) and from plans to achieve the goal (in the case of goals). We give formal definitions of necessary and contingent conditions for actions, plans, and goals in the sections below.

3.1 Actions

Actions are the basic steps an agent can perform in order to change its environment. Actions may be deterministic or non-deterministic. Deterministic actions have a single outcome (postcondition), while the execution of a non-deterministic action results in one of a set of possible outcomes (set of postconditions).

The precondition of an action α, $pre_n(\alpha) = \phi$, is always necessary (and $pre_c(\alpha) = \emptyset$). The in-condition of an action, $in_n(\alpha) = \upsilon$, is also necessary (and $in_c(\alpha) = \emptyset$). Deterministic actions have a single postcondition ψ. The necessary postcondition of a deterministic action α is defined as $post_n(\alpha) = \psi$, and the contingent postcondition is defined by $post_c(\alpha) = \emptyset$. The execution

of a non-deterministic action results in one of a set of possible postconditions $\{\psi_1, \ldots, \psi_n\}, \psi_i \subseteq \Phi^+$. The necessary postcondition of a non-deterministic action α is defined as $post_n(\alpha) = \bigcap \psi_i \in \{\psi_1, \ldots, \psi_n\}$, and the contingent postcondition is defined by $post_c(\alpha) = \bigcup \psi_i \in \{\psi_1, \ldots, \psi_n\} \setminus post_n(\alpha)$.[6] We assume that action specifications are consistent in the sense that each possible outcome of an action is itself consistent, i.e., that $\psi_i \not\models \bot, 1 \leq i \leq n$, and that execution of an action does not invalidate the in-condition of the action, i.e., $in_n(\alpha) \cup post_n(\alpha) \cup post_c(\alpha) \not\models \bot$.[7]

3.2 Plans

A plan π consists of a sequence of actions, subgoals, and parallel compositions of actions and subgoals. That is, a plan is of the form $\pi = \alpha_1; \ldots; \alpha_m$, where each α_i is either an action, a subgoal or a parallel composition $\beta_1 \| \ldots \| \beta_k$, where each β_i is either an action or a subgoal. In the interests of generality, we make no assumptions about the execution of a parallel composition of actions and subgoals: steps $\beta_1, \ldots, \beta_k$ may be executed in parallel, i.e., they may overlap in any of the ways defined in [1], or their execution may be arbitrarily interleaved.[8] However, we require that there are no conflicts between the pre-, in- and postconditions of $\beta_1, \ldots, \beta_k$ and, as a result, the overall postcondition of the parallel composition is "stable", i.e., for each $\beta_i, \beta_j, 1 \leq i \leq k, 1 \leq j \leq k, i \neq j$, the necessary and contingent postconditions of β_i must be consistent with the necessary and contingent pre- in- and postconditions β_j.

We define the necessary and contingent conditions of a parallel composition as follows. The necessary postcondition of a parallel composition $\alpha = \beta_1 \| \ldots \| \beta_k$ is defined as

$$post_n(\alpha) = \bigcup_{i=1}^{k} post_n(\beta_i).$$

The contingent postcondition of a parallel composition is defined similarly, except that we exclude any contingent postcondition literal of a step if it is also a necessary postcondition of some other step, i.e.,

$$post_c(\alpha) = \bigcup_{i=1}^{k} post_c(\beta_i) \setminus post_n(\alpha).$$

However, the necessary pre- and in-conditions of a parallel composition need to take into account postconditions of steps that possibly establish by virtue

[6] Note that this defines the contingent conditions of an action as distinct from the necessary conditions.

[7] For entailment, we sometimes treat a conjunction of literals as a set consisting of the literals.

[8] For example, if β_i is an action and β_j a subgoal, then β_i may be interleaved with the actions appearing in the goal-plan tree for β_j.

of how steps are interleaved or overlapped the pre- and in-conditions of other steps, i.e.,

$$con_n(\alpha) = \bigcup_{i\in\{1,\ldots,k\}} \big(con_n(\beta_i) \setminus \bigcup_{j\in\{1,\ldots,k\}\setminus\{i\}} (post_n(\beta_j) \cup post_c(\beta_j))\big),$$

where *con* is either *pre* or *in*. Finally, the contingent pre- and in-conditions of a parallel composition is defined as

$$con_c(\alpha) = \bigcup_{i=1}^{k} \big(con_c(\beta_i) \cup con_n(\beta_i)\big) \setminus con_n(\alpha)$$

where *con* is either *pre* or *in*.

We can now define the necessary and contingent pre-, in- and postconditions of plans. The necessary precondition of a plan $\pi = \alpha_1; \ldots; \alpha_m$ is defined as

$$pre_n(\pi) = pre_n(\alpha_1) \cup \bigcup_{i=2}^{m} \left[pre_n(\alpha_i) \setminus \bigcup_{j=1}^{i-1} post_n(\alpha_j) \cup post_c(\alpha_j) \right]$$

that is, the necessary preconditions of steps that are not established by the necessary or contingent postconditions of previous steps. Necessary preconditions must hold for all executions of π.[9] Note that we do not assume that a plan establishes all the preconditions of the steps in the plan. For example, a plan to make coffee may assume that the agent is in the kitchen and that there is coffee in the kitchen. However, we do assume that each plan π ensures a free-choice' among its descendant' plans (plans that achieve the subgoals of π). For example, a plan to make coffee should not cause the agent to leave the kitchen before the coffee is made, as that would then invalidate one or more subplans, e.g. one that grinds the coffee. More precisely, for any step α_k in a plan $\pi = \alpha_1; \ldots; \alpha_n$, if there is an earlier step α_i $(i < k)$ and a literal $l \in post_n(\alpha_i)$ such that $\sim l \in pre_n(\alpha_k) \cup pre_c(\alpha_k) \cup in_n(\alpha_k) \cup in_c(\alpha_k)$, then there is also an intermediate step α_j $(i < j < k)$ with $\sim l \in post_n(\alpha_j) \cup post_c(\alpha_j)$, where $\sim l = \neg p$ if $l = p$ and $\sim l = p$ if $l = \neg p$.

If π contains non-deterministic actions or subgoals, it may also have contingent preconditions, i.e., preconditions which may have to be established depending on the outcome of a non-deterministic action (if the outcome of the action fails to achieve the precondition of a later action in the plan) or the choice of plan to achieve a subgoal. Thus, the contingent precondition of a plan $\pi = \alpha_1; \ldots; \alpha_m$ is defined as $pre_c(\pi) = pre_c(\alpha_1) \cup$

$$\bigcup_{i=2}^{m} \left[\Big(pre_c(\alpha_i) \setminus \bigcup_{j=1}^{i-1} post_n(\alpha_j) \Big) \cup \Big(\big(pre_n(\alpha_i) \setminus \bigcup_{j=1}^{i-1} post_n(\alpha_j) \big) \cap \bigcup_{j=1}^{i-1} post_c(\alpha_j) \Big) \right].$$

[9] As we are concerned with the executability of plans rather than their applicability in a particular context, we do not include the context condition (belief context) of a plan specified by a developer to be part of its precondition. However, in a well-formed plan, the necessary precondition should form (part of) the context condition of the plan.

That is, the possible preconditions of each step not established by necessary postconditions of previous steps, and the necessary preconditions of each step that are (possibly) established by contingent postconditions of previous steps, but not by their necessary postconditions. Observe that sets $pre_n(\pi)$ and $pre_c(\pi)$ are mutually exclusive by definition.

The necessary in-condition of a plan $\pi = \alpha_1; \ldots; \alpha_m$ is defined as

$$in_n(\pi) = \bigcup_{i=1}^{m-1} (in_n(\alpha_i) \cap in_n(\alpha_{i+1})).$$

That is, a necessary in-condition of a plan π is an in-condition that is common across two or more consecutive steps in π.

The contingent in-condition of a plan $\pi = \alpha_1; \ldots; \alpha_m$ is defined as

$$in_c(\pi) = \bigcup_{i=1}^{m} \big(in_c(\alpha_i) \cup in_n(\alpha_i)\big) \setminus in_n(\pi).$$

Finally, we define the necessary and contingent postconditions of a plan. The necessary postconditions of a plan $\pi = \alpha_1; \ldots; \alpha_m$ is defined as

$$post_n(\pi) = \{l \mid \exists i : l \in post_n(\alpha_i) \wedge \forall j \in \{i, \ldots, m\} :\sim l \notin post_n(\alpha_j) \cup post_c(\alpha_j)\}.$$

That is, the necessary postconditions of each step not 'undone' by the necessary or contingent postconditions of later steps. The contingent postcondition of a plan $\pi = \alpha_1; \ldots; \alpha_m$ is defined as $post_c(\pi) = post_c^1(\pi) \cup post_c^2(\pi)$, where

$$post_c^1(\pi) = \{l \mid \exists i : l \in post_c(\alpha_i) \wedge \forall j \in \{i, \ldots, m\} :\sim l \notin post_n(\alpha_j)\}$$

and

$$\begin{aligned} post_c^2(\pi) = \{l \mid \exists i : l \in post_n(\alpha_i) \wedge \\ \exists j \in \{i, \ldots, m\} :\sim l \in post_c(\alpha_j) \wedge \\ \forall j \in \{i, \ldots, m\} :\sim l \notin post_n(\alpha_j)\}. \end{aligned}$$

That is, the contingent postcondition of a plan is either a contingent postcondition of a step that is not 'undone' by the necessary postcondition of a later step, or a necessary postcondition of a step that may be 'undone' by a contingent postcondition of a later step. Observe that sets $post_n(\pi)$ and $post_c(\pi)$ are mutually exclusive by definition.

3.3 Goals

A goal γ is associated with a set of plans $\pi_1, \ldots, \pi_n$ that achieve γ, and the pre-, in- and postconditions of γ are derived from this set of associated plans. For simplicity, we stipulate that goals with the same *GoalType* as γ do not appear in the goal-plan tree rooted at γ.[10]

[10] This is a standard assumption in computing summary information e.g., [7, 8, 16, 15]. The assumption can be relaxed, but the definitions of conditions below become more complex.

The necessary pre-, in- and postconditions of a goal γ associated with plans $\pi_1, \ldots, \pi_n$ is defined as

$$con_n(\gamma) = \bigcap_{i=1}^{n} con_n(\pi_i),$$

where *con* is either *pre*, *in* or *post*. That is, necessary pre-, in-, or postconditions must hold respectively before, during, or after all ways of achieving γ.

The contingent pre-, in-, and postconditions of a goal γ associated with plans $\pi_1, \ldots, \pi_n$ is defined as

$$con_c(\gamma) = \bigcup_{i=1}^{n} con_c(\pi_i) \cup \left[\bigcup_{j=1}^{n} con_n(\pi_j) \setminus con_n(\gamma) \right]$$

where *con* is either *pre*, *in* or *post*. That is, a pre-, in-, or postcondition is contingent for γ if it is a contingent condition of a plan π_i to achieve γ, or if it is a necessary condition for a plan π_j but not for γ itself (i.e., it is a necessary condition of some but not all plans for γ.)

The definitions above capture the relationship between the pre-, in- and postconditions of actions, plans and goals in a goal-plan tree. The conditions for actions define which propositions must be true before, during and after either all executions of an action (necessary conditions), or some execution of the action (contingent conditions). The conditions for plans define which propositions must be true before, during and after either all executions of a plan, or some execution of the plan. The necessary preconditions of a plan specify the states in which the plan is applicable. The conditions for goals define which propositions must be true before, during and after either all means of achieving a goal or some means of achieving a goal.

4 Execution Conditions

In the previous section, we defined the necessary and contingent conditions for the execution of a single goal-plan tree. In this section, we consider information relevant to the execution of a *set* of goal plan trees.

If an agent always executes at most one goal-plan tree at a time, e.g., it executes its intentions in first-in-first-out order, then the execution conditions are the same as those given in Section 3. However, in many application domains, the goal-plan trees comprising a system or an agent's user supplied domain knowledge are executed in parallel. For example, in many BDI agent architectures, the plans comprising the agent's intentions are executed in parallel, e.g., by executing one step of an intention at each cycle in a round robin fashion [4, 19]. Interactions between interleaved steps in plans in different goal-plan trees may result in conflicts, i.e., the execution of a step in one plan makes the execution of a step in another concurrently executing plan impossible. Given a set of

goal-plan trees, the *scheduling problem* is to determine which step of which goal-plan tree to execute next, so as to minimise the number of execution conflicts.[11] Scheduling aims to minimise the number of plan failures resulting from choices made by the agent regarding the order of execution of a set of goal-plan trees, thus allowing the largest number of goals to be achieved.[12] Our aim is not to solve the scheduling problem here; for example, we do not consider the problem of which plan an agent should adopt for a given (sub)goal this is the concern of deliberation scheduling. Rather, we focus on defining conditions that must or may hold on all possible future executions of a set of goal-plan trees. As such, the conditions we define should be taken into account by any scheduler, but are neutral with respect to the actual form of deliberation scheduling adopted. It turns out that, in our setting, the information relevant for scheduling differs from the conditions on the well-formedness of a goal-plan tree defined in the previous section. As such, the definitions below depart from those in, e.g., [15, 14].

To define the execution conditions for a set of goal-plan trees, we need some auxiliary notions. Given a set of goal-plan trees $T = \{\tau_1, \ldots, \tau_n\}$, an *execution context* for T is a set of pairs $I = \{(\tau_1, \rho_1), \ldots, (\tau_n, \rho_n)\}$, where each ρ_i defines the set of possible future execution paths for τ_i. Each ρ_i corresponds to the point execution has reached in the goal-plan tree τ_i, and hence the possible paths future execution of τ_i may follow. (I essentially corresponds to the intentions of a BDI agent.) Initially, each ρ_i points to the top-level goal of the corresponding goal-plan tree τ_i. As execution of τ_i proceeds, plans are selected, restricting the possible future execution paths to a subtree of τ_i captured by ρ_i. In the interests of brevity, and where no confusion can arise, we shall refer to possible future execution paths simply as possible execution paths.

An initial set of possible execution paths ρ_0 for a goal-plan tree τ is a sequence $(\pi_i, \alpha_1), \ldots, (\pi_i, \alpha_k)$, where $\pi_i = \alpha_1; \ldots; \alpha_k$ is the selected plan for the top-level goal of τ. As execution progresses, a set of possible execution paths $\rho = (\pi_1, \alpha_1), (\pi_2, \alpha_2), \ldots, (\pi_m, \alpha_m)$ evolves as follows. The successor set of possible execution paths ρ' of ρ is $(\pi_2, \alpha_2), \ldots, (\pi_m, \alpha_m)$ if α_1 is an action, and $\rho' = (\pi'_1, \alpha'_1), \ldots, (\pi'_1, \alpha'_n), (\pi_2, \alpha_2), \ldots, (\pi_m, \alpha_m)$ if α_1 is a subgoal γ_1 and $\pi'_1 = \alpha'_1; \ldots; \alpha'_n$ is the plan selected for γ_1. Only sets of possible execution paths which are the initial set of possible execution paths in τ (corresponding to the top-level of goal of τ) or are obtained by the progression step described above are sets of possible execution paths in τ.

We can now define the necessary and contingent execution conditions of an execution context. Informally, the necessary execution conditions of a set of

[11] Scheduling may also be used to maximise the number of positive interactions between goal-plan trees, as in, e.g., [16, 2]; we do not consider positive interactions here.

[12] Plans may fail for reasons that are outside the control of the agent, e.g., due to changes in the environment, or actions of other agents violating the conditions of a plan. Several approaches, e.g., [17, 18, 20] have been proposed which attempt to avoid such failures. However, the information about goal-plan trees required by these approaches (essentially the the percentage of world states for which there is some applicable plan for any subgoal within an intention) is different from that required for scheduling, and we do not consider them further here.

possible execution paths ρ_i, are those conditions that must hold or be achieved at some point in all possible future executions of a goal-plan tree τ_i starting from ρ_i, and the contingent execution conditions are those conditions that must hold or be achieved at some point of time in at least one possible future execution (but not all executions) of τ_i starting from ρ_i. When executing the set of goal-plan trees in T in parallel, such execution conditions must be protected if the execution conditions of two sets of possible execution paths ρ_i and ρ_j intersect, then interleaving steps in ρ_i and ρ_j may result in conflicts.

4.1 Actions

As actions are atomic, the necessary and contingent execution conditions of an action α are identical to the corresponding necessary and contingent conditions for α (we denote execution conditions with a $*$):

$$con^*_n(\alpha) = con_n(\alpha) \qquad con^*_c(\alpha) = con_c(\alpha)$$

where con^*_n and con_n are either pre^*_n and pre_n, in^*_n and in_n or $post^*_n$ and $post_n$ respectively, and similarly con^*_c and con_c are either pre^*_c and pre_c, in^*_c and in_c or $post^*_c$ and $post_c$.

4.2 Plans

The necessary and contingent execution conditions of a plan π differ from the corresponding necessary and contingent conditions for π. As steps in plans in different goal-plan trees may be arbitrarily interleaved, we need to protect *all* the preconditions in a plan, even if they are established by a preceding step in the same plan, as the condition may be invalidated by a step in a plan in another goal-plan tree.

The necessary execution pre-, in- and postcondition of a parallel composition $\alpha = \beta_1 \| \ldots \| \beta_k$ is the union of the necessary execution conditions of each β_i, i.e.,

$$con^*_n(\alpha) = \bigcup_{i=1}^{k} con^*_n(\beta_i)$$

where con^*_n is either pre^*_n, in^*_n or $post^*_n$.

The contingent pre-, in- and post- execution conditions of a parallel composition is also defined as the union of contingent execution conditions of each β_i, except that we exclude any contingent postcondition literal of a step if it is also a necessary postcondition of some other step, i.e.,

$$con^*_c(\alpha) = \bigcup_{i=1}^{k} con^*_c(\beta_i) \setminus con^*_n(\alpha).$$

The necessary and contingent execution preconditions of a plan (or plan suffix) $\pi = \alpha_1; \ldots; \alpha_m$ are therefore given by

$$pre^*_n(\pi) = \bigcup_{i=1}^{m} pre^*_n(\alpha_i) \qquad pre^*_c(\pi) = \bigcup_{i=1}^{m} pre^*_c(\alpha_i) \setminus pre^*_n(\pi).$$

Similarly, the postconditions of interest are no longer the 'eventual' postconditions of the plan, since the postcondition of an action α_i 'undone' by a later step $\alpha_j, i < j$ in π may be 'visible' to a step in a plan in another goal-plan tree. The necessary and contingent execution postconditions of π are therefore given by

$$post_n^*(\pi) = \bigcup_{i=1}^{m} post_n^*(\alpha_i) \qquad post_c^*(\pi) = \bigcup_{i=1}^{m} post_c^*(\alpha_i) \setminus post_n^*(\pi).$$

In contrast, the necessary and contingent execution in-conditions of π are the same as the necessary and contingent in-conditions of π: $in_n^*(\pi) = in_n(\pi)$, $in_c^*(\pi) = in_c(\pi)$. (Since $in_n(\pi)$ and $in_c(\pi)$ define conditions that must hold between the execution of steps in π, they are unaffected by interleaving of plan steps.)

4.3 Goals

As with plans, the necessary and contingent execution conditions of a goal γ associated with plans $\pi_1, \ldots, \pi_n$ differ from the corresponding necessary and contingent conditions for γ. (The conditions of goals are defined in terms of the conditions of their associated plans.)

The necessary pre-, in- and post- execution conditions of a goal γ associated with plans $\pi_1, \ldots, \pi_n$ is defined as

$$con_n^*(\gamma) = \bigcap_{i=1}^{n} con_n^*(\pi_i),$$

where *con* is either *pre*, *in* or *post*. That is, necessary pre-, in-, or post- execution conditions must hold respectively before, during, or after all ways of achieving γ.

The contingent pre-, in-, and post- execution conditions of a goal γ associated with plans $\pi_1, \ldots, \pi_n$ is defined as

$$con_c^*(\gamma) = \bigcup_{i=1}^{n} con_c^*(\pi_i) \cup \left[\bigcup_{j=1}^{n} con_n^*(\pi_j) \setminus con_n^*(\gamma) \right],$$

where *con* is either *pre*, *in* or *post*. That is, a pre-, in-, or postcondition is contingent for γ if it is a contingent condition of a plan π_i to achieve γ, or if it is a necessary condition for a plan π_j but not for γ itself (i.e., it is a necessary condition of some but not all plans for γ.)

4.4 Sets of Execution Paths

We can now define the necessary and contingent execution conditions of a set of possible execution paths $\rho = (\pi_1, \alpha_1), \ldots, (\pi_k, \alpha_k)$ of a goal-plan tree τ.

The necessary execution precondition of a set of possible execution paths ρ is given by

$$pre_n^*(\rho) = \bigcup_{i=1}^{k} pre_n^*(\alpha_i).$$

That is, we must protect the necessary preconditions of all steps in ρ. The contingent execution precondition of ρ is given by

$$pre_c^*(\rho) = \bigcup_{i=1}^{k} pre_c^*(\alpha_i) \setminus pre_n^*(\rho).$$

Contingent preconditions are those that may need to be established during execution, depending on the choice of plan to achieve a goal.

The necessary execution in-condition of a set of possible execution paths ρ is given by

$$in_n^*(\rho) = \bigcup_{i=1}^{k} in_n^*(\alpha_i) \cup \bigcup_{i=1}^{k} in_n(\pi_i).$$

That is, we must protect the in-conditions of all steps in ρ, and in addition we also need to protect the in-conditions of all currently executing plans in ρ. The contingent execution in-condition of ρ is given by

$$in_c^*(\rho) = \bigcup_{i=1}^{k} in_c^*(\alpha_i) \cup \bigcup_{i=1}^{k} in_c(\pi_i) \setminus in_n^*(\rho).$$

The necessary and contingent execution postconditions of a set of possible execution paths ρ is given by

$$post_n^*(\rho) = \bigcup_{i=1}^{k} post_n^*(\alpha_i) \qquad post_c^*(\rho) = \bigcup_{i=1}^{k} post_c^*(\alpha_i) \setminus post_n^*(\rho).$$

Finally, the necessary execution conditions of a set of possible execution paths ρ are given by

$$cond_n^*(\rho) = pre_n^*(\rho) \cup in_n^*(\rho) \cup post_n^*(\rho),$$

and the contingent execution conditions of ρ are given by

$$cond_c^*(\rho) = pre_c^*(\rho) \cup in_c^*(\rho) \cup post_c^*(\rho).$$

Conflicts may occur when we have complementary literals in the execution conditions of two sets of possible execution paths, ρ_i and ρ_j, i.e., when

$$\exists\, l \in cond_x^*(\rho_i) \wedge \sim l \in cond_x^*(\rho_j),$$

where $cond_x^*$ is either $cond_n^*$ or $cond_c^*$. Clearly, there are different cases. For example, conflicts between the necessary execution conditions of two execution

paths may be a more serious problem than conflicts between contingent execution conditions.

If no conflicts (as defined above) occur between two sets of possible execution paths ρ_i and ρ_j, then the next step in either (or both) ρ_i and ρ_j may be safely executed. On the other hand, if there are conflicts between the two sets of possible execution paths, then we could still interleave their execution such that they do not interfere with one another, e.g., by borrowing techniques from [15]. For example, if the conflict between ρ_i and ρ_j is due to complementary literals in $in_n^*(\rho_i)$ and $post_n^*(\rho_j)$, then we could delay the execution of ρ_j until ρ_i progresses to a point where there is no longer a conflict with ρ_j. This is because ρ_j might otherwise interfere with the in-condition of a plan that is currently being pursued.[13] If the conflict between ρ_i and ρ_j is due to complementary literals in $pre_c^*(\rho_i)$ and $post_c^*(\rho_j)$, an optimistic approach would be to first execute ρ_j until it progresses to a point where a conflict no longer occurs with ρ_i, and only then begin executing ρ_i. This assumes that either execution of ρ_j does not actually bring about the conflicting literal, or that if it is brought about, execution of ρ_i is such that the conflicting contingent precondition is not required, or a step within ρ_i itself asserts the negation of the conflicting literal.

5 Conclusion and Future Work

This paper has provided definitions of pre-, in-, and postconditions of actions, plans, and goals, for an extended goal-plan tree that supports the execution of steps (goals and actions) in parallel, as well as the specification of both deterministic and non-deterministic actions. Our definitions essentially capture 'static' and 'dynamic' notions of conditions, which are derived from the primitive ones specified within the basic actions that form plans. We believe that 'static' properties defined by our notions will facilitate authoring agent programs, particularly because it is important to know the properties of the individual "building blocks" (goal-plan trees) that are available when composing a new plan. We have used our 'dynamic' notions of conditions to derive the conditions that must be protected by any scheduler, when interleaving two or more goal-plan trees.

We foresee two main directions for future work. First, we could allow for a step in a plan to necessarily invalidate one or more (though not all) 'descendant' (sub)plans of a later step, and accordingly extend our notions of the necessary and contingent postconditions of a plan. This extension would involve identifying which postconditions in the later step are never asserted due to the conflict (and are thereby neither necessary nor contingent postconditions), and which ones

[13] Note that if ρ_i and ρ_j are considered in order of the priority of the associated top-level goal (or ties are broken arbitrarily), deadlock (as defined in [15, 14]) cannot arise, even if there are complementary literals in $in_n^*(\rho_j)$ and $post_n^*(\rho_i)$. However, this may result in conditions of the lower priority set of possible execution paths being violated. In such cases, more sophisticated intention scheduling techniques, e.g., [21, 20] may be able to find an interleaving that protects the conditions of both sets of possible execution paths.

are always asserted due to the conflict, by virtue of certain descendant plans being always inapplicable. Second, we could explore how to generate a schedule for interleaving two or more goal-plan trees, while respecting the conditions that we have identified as needing to be protected.

References

1. James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.
2. Yuan Yao an Brian Logan and John Thangarajah. Robust Execution of BDI Agent Programs by Exploiting Synergies Between Intentions. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, page to appear, 2016.
3. Jorge A. Baier, Christian Fritz, and Sheila A. McIlraith. Exploiting procedural domain control knowledge in state-of-the-art planners. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 26–33, 2007.
4. Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. Wiley Series in Agent Technology. Wiley, 2007.
5. Adi Botea, Markus Enzenberger, Martin Müller, and Jonathan Schaeffer. Macro-FF: Improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research (JAIR)*, 24:581–621, 2005.
6. David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377, 1987.
7. Bradley J. Clement and Edmund H. Durfee. Theory for coordinating concurrent hierarchical planning agents using summary information. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 495–502, 1999.
8. Bradley J. Clement, Edmund H. Durfee, and Anthony C. Barrett. Abstract reasoning for planning and coordination. *Journal of Artificial Intelligence Research (JAIR)*, 28:453–515, 2007.
9. Lavindra de Silva, Sebastian Sardina, and Lin Padgham. First Principles Planning in BDI systems. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1105–1112, 2009.
10. Kutluhan Erol, James Hendler, and Dana S. Nau. HTN planning: Complexity and expressivity. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 1123–1128, 1994.
11. Christian Fritz, Jorge A. Baier, and Sheila A. McIlraith. ConGolog, Sin Trans: Compiling ConGolog into Basic Action Theories for planning and beyond. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 600–610, 2008.
12. Subbarao Kambhampati, Amol Dattatraya Mali, and Biplav Srivastava. Hybrid planning for partially hierarchical domains. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 882–888, 1998.
13. Sebastian Sardina, Lavindra de Silva, and Lin Padgham. Hierarchical planning in BDI agent programming languages: A formal approach. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1001–1008, 2006.
14. John Thangarajah and Lin Padgham. Computationally effective reasoning about goal interactions. *Journal of Automated Reasoning*, 47(1):17–56, 2011.

15. John Thangarajah, Lin Padgham, and Michael Winikoff. Detecting and avoiding interference between goals in intelligent agents. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 721–726, 2003.
16. John Thangarajah, Lin Padgham, and Michael Winikoff. Detecting and exploiting positive goal interaction in intelligent agents. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 401–408, 2003.
17. John Thangarajah, Sebastian Sardina, and Lin Padgham. Measuring plan coverage and overlap for agent reasoning. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1049–1056, 2012.
18. Max Waters, Lin Padgham, and Sebastian Sardina. Evaluating coverage based intention selection. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS)*, pages 957–964, 2014.
19. Michael Winikoff. JACK Intelligent Agents: An Industrial Strength Platform. In *Multi-Agent Programming*, pages 175–193. Springer, 2005.
20. Yuan Yao and Brian Logan. Action-level intention selection for BDI agents. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2016. (to appear).
21. Yuan Yao, Brian Logan, and John Thangarajah. SP-MCTS-based intention scheduling for BDI agents. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 1133–1134, 2014.

Keyword Index

Author Index