Lecture Notes in Artificial Intelligence 3529

Edited by J. G. Carbonell and J. Siekmann

Subseries of Lecture Notes in Computer Science

Manuel Kolp Paolo Bresciani
Brian Henderson-Sellers Michael Winikoff (Eds.)

Agent-Oriented Information Systems III

7th International Bi-Conference Workshop, AOIS 2005
Utrecht, Netherlands, July 26, 2005
and Klagenfurt, Austria, October 27, 2005
Revised Selected Papers

Springer

Series Editors

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

Volume Editors

Manuel Kolp
Catholic University of Louvain (UCL)
School of Management (IAG), Information Systems Research Unit (ISYS)
1, Place des Doyens, 1348 Louvain-La-Neuve, Belgium
E-mail: kolp@isys.ucl.ac.be

Paolo Bresciani
European Commission
DG Information Society and Media, Unit D3: Software Technologies
Avenue de Beaulieu 29, level 4, office 49, 1049 Brussels, Belgium
E-mail: paolo.bresciani@ec.europa.eu

Brian Henderson-Sellers
University of Technology, Sydney
Faculty of Information Technology
P.O. Box 123, Broadway, NSW 2007, Australia
E-mail: brian@it.uts.edu.au

Michael Winikoff
RMIT University
School of Computer Science and Information Technology
Melbourne, VIC 3001, Australia
E-mail: winikoff@cs.rmit.edu.au

Library of Congress Control Number: 2006936083

CR Subject Classification (1998): I.2.11, H.4, H.3, H.5.2-3, C.2.4, I.2

LNCS Sublibrary: SL 7 – Artificial Intelligence

ISSN 0302-9743
ISBN-10 3-540-48291-1 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-48291-8 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11916291 06/3142 5 4 3 2 1 0

Preface

Information systems underpin today's business and entertainment. The means by which these information systems have been developed has changed over the years. Although the current paradigm is to use object-oriented concepts, a new set of concepts, focussed on agent technology, is starting to be evaluated. Agents offer higher level abstractions (than objects) for the conceptualization, design and implementation of information systems. Agents have autonomy, can reason and can coordinate within societies of agents.

The AOIS series of workshops explores the potential for facilitating the increased usage of agent technology in the creation of information systems in the widest sense. In 2005, two AOIS workshops were held internationally. The first was affiliated with the AAMAS 2005 meeting in July in Utrecht in The Netherlands and chaired by Henderson-Sellers and Winikoff and the second with ER 2005 in November in Klagenfurt in Austria and chaired by Kolp and Bresciani. The best papers from these meetings were identified and authors invited to revise and possibly extend their papers in the light of reviewers' comments and feedback at the workshop.

We have grouped these papers loosely under four headings: Agent behavior, communications and reasoning; Methodologies and ontologies; Agent-oriented software engineering; and Applications. These categories fairly represent the breadth of current AOIS research as well as encompassing the papers presented at the two AOIS workshops. We trust you will find the content of these selected and revised papers to be of interest and utility.

Since the papers presented at the Utrecht workshop were not formally published, some of the authors chose not to make any significant extension to their papers. On the other hand, the Klagenfurt workshop papers were published by Springer as part of the ER proceedings and thus have been significantly extended before acceptance for this volume. All invited papers for this volume were re-reviewed (in their extended forms) by three members of the Program Committee prior to acceptance. We wish to thank all authors for undertaking the necessary revisions and meeting the editorial deadlines.

September 2006

Manuel Kolp
Paolo Bresciani
Brian Henderson-Sellers
Michael Winikoff

Organization

Workshop Co-chairs

Manuel Kolp (Catholic University of Louvain, Belgium)
Paolo Bresciani (IRST-ITC, Italy)
Brian Henderson-Sellers (University of Technology, Sydney, Australia)
Michael Winikoff (RMIT, Australia)

Steering Committee

Yves Lesperance (York University, Canada)
Gerd Wagner (Eindhoven University of Technology, Netherlands)
Eric Yu (University of Toronto, Canada)
Paolo Giorgini (University of Trento, Italy)

Program Committee

Carole Bernon (University Paul Sabatier, Toulouse, France)
Brian Blake (Georgetown University, Washington DC, USA)
Paolo Bresciani (ITC-IRST, Italy)
Jaelson Castro (Federal University of Pernambuco, Brazil)
Luca Cernuzzi (Universidad Católica Nuestra Señora de la Asunción, Paraguay)
Massimo Cossentino (ICAR-CNR, Palermo, Italy)
Luiz Cysneiros (York University, Toronto)
John Debenham (University of Technology, Sydney)
Scott DeLoach (Kansas State University, USA)
Frank Dignum (University of Utrecht, Netherlands)
Paolo Donzelli (University of Maryland, College Park, USA)
Bernard Espinasse (Domaine Universitaire de Saint-Jérôme, France)
Stéphane Faulkner (University of Namur, Belgium)
Behrouz Homayoun Far (University of Calgary, Canada)
Innes Ferguson (B2B Machines, USA)
Alessandro Garcia (PUC Rio)
Chiara Ghidini (ITC-IRST, Italy)
Aditya Ghose (University of Wollongong, Australia)
Marie-Paule Gleizes (University Paul Sabatier, Toulouse, France)
Cesar Gonzalez-Perez (University of Technology, Sydney, Australia)
Giancarlo Guizzardi (University of Twente, Netherlands)
Igor Hawryszkiewycz (University of Technology, Sydney, Australia)
Brian Henderson-Sellers (University of Technology, Sydney, Australia)
Carlos Iglesias (Technical University of Madrid, Spain)

Manuel Kolp (Catholic University of Louvain, Belgium)
Daniel E. O'Leary (University of Southern California, USA)
Carlos Lucena (PUC Rio, Brazil)
Graham Low (UNSW, Australia)
Philippe Massonet (CETIC, Belgium)
Haris Mouratidis (University of East London, UK)
Jörg Mueller (Siemens, Germany)
Juan Pavón (Universidad Complutense Madrid, Spain)
Omer F. Rana (Cardiff University, UK)
Onn Shehory (IBM Haifa Labs, Israel)
Nick Szirbik (Technische Universiteit Eindhoven, Netherlands)
Kuldar Taveter (University of Melbourne, Australia)
Quynh-Nhu Numi Tran (UNSW, Australia)
Viviane Torres da Silva (PUC Rio, Brazil)
Michael Winikoff (RMIT, Australia)
Carson Woo (University of British Columbia, Canada)
Bin Yu (North Carolina State University, USA)
Amir Zeid (American University of Cairo, Egypt)
Zili Zhang (Deakin University, Australia)

Table of Contents

Agent Behavior, Communications and Reasoning

Methodologies and Ontologies

Agent-Oriented Software Engineering

Applications

Automated Interpretation of Agent Behaviour

D.N. Lam and K.S. Barber

The University of Texas at Austin
The Laboratory for Intelligent Processes and Systems
dnlam@lips.utexas.edu, barber@lips.utexas.edu

Abstract. Software comprehension, which is essential for debugging and maintaining software systems, has lacked attention in the agent community. Comprehension has been a manual process, involving the analysis and interpretation of log files that record agent behaviour in the implemented system. This paper describes an approach and tool to automate creating interpretations of agent behaviour from observations of the implementation execution, thus helping users (i.e. designers, developers, and end-users) to understand the motivations of agent actions. By explicitly modelling the user's comprehension of the implemented system as background knowledge for the tool, feedback can be provided as to whether the user's comprehension accurately represents the implementation's behaviour and, if not, how it can be corrected. Additionally, with the aid of the Tracer Tool, many of the manual tasks are automated, such as verifying that agents are behaving as expected, identifying unexpected behaviour and generating explanations for any particular observation.

1 Introduction

Agents are distributed software entities that are capable of autonomous decision-making. Besides being motivated by its own goals, an agent's behaviour is influenced by interactions with other agents (i.e. their goals, beliefs and intentions), by events that have occurred in the past and by the current situation. With so many factors that can influence an agent's decision, end-users may not trust the agent's decision, and developers may have difficulty debugging the implementation. Software designers, developers and end-users often need to comprehend why an agent acted in a particular way when situated in its operating environment, which itself can be unpredictable and uncertain. Currently, the process of comprehending agent behaviour is done manually by *interpreting* the observations from the implementation executions to create a connected, comprehensive view of what the software is doing. The interpretation process links (usually with a causal link) actual observations together using the user's comprehension (or background knowledge of expected behaviour). In essence, an interpretation compares the actual implementation behaviour with expected behaviour, which may have been created by the user from the software design, previous experience, intuition etc.

Considering the complexities of agent software (e.g. autonomous decision-making and a high degree of interaction) and the usual disparity between

M. Kolp et al. (Eds.): AOIS 2005, LNAI 3529, pp. 1–15, 2006.

software design and implementation, software comprehension is a difficult, time-consuming and tedious process. To alleviate these issues, this research aims to automate the comprehension process as much as possible. This paper describes (1) how the Tracer Tool can be used to help build and verify a model of the user's comprehension of the implemented agent system's behaviour (i.e. background knowledge) and (2) how an interpretation of agent behaviour can be automatically generated from the background knowledge and recorded observations.

Sophisticated software such as an agent-based system presents obstacles that are difficult to overcome using current software comprehension and verification tools. In general, traditional software comprehension (or reverse engineering) tools are limited by their low abstraction level, their dependence on analyzing source code, their lack of automation to help decipher tremendous amounts of collected data and their lack of a model for how much the user understands. Taking the formal approach to modelling systems, model-checking facilitates comprehension by verifying properties of systems but is limited by its demand for expert knowledge of the model-checking process, its high computational complexity, and the translation gap between the model being checked and the actual system.

To remedy limitations of current comprehension techniques, this research offers a novel approach to computer-aided software comprehension that involves: (1) modelling the user's comprehension of the system as background knowledge usable by tools, (2) ensuring that the user's comprehension accurately reflects the actual system and (3) generating interpretations and explanations as evidence of comprehension.

This paper describes an approach and tool that builds on the ideas from reverse engineering and model-checking to better assist the human user (of various skill levels) in comprehending agent-based software. Section 2 reviews limitations of existing work and highlights advantages that are used in this research. Section 3 presents the formulation of the problem and the approach employed to automate building the interpretation of agent behaviour. Section 4 describes how the Tracer Tool implements the approach. Section 5 demonstrates how the interpretation can be used to generate explanations. Finally, Section 6 summarizes the contributions of this research.

2 Background

Agent concepts (i.e. beliefs, goals, intentions, actions, events and messages) are abstractions of low-level implementation constructs (e.g. data structures, classes and variables) that make designing and communicating the design easier. Though agent concepts help in designing software for sophisticated and distributed domains, there has been little research in leveraging them for the expensive maintenance phase of software engineering. Since software designs use agent concepts to describe agent structure (e.g. an agent encapsulates localized beliefs, goals, and intentions) and behaviour (e.g. an agent performs an action when it believes an event occurred), agent concepts should be leveraged for comprehending the

software. If the same concepts and models are used in forward and reverse engineering, tools would be able to better support re-engineering, round-trip engineering, maintenance and reuse [1]. In this research, agent concepts are used to take advantage of the user's intuitive knowledge of agent-based systems to comprehend agent behaviour in the implementation. The set of concepts can be extended or replaced by practically any set of concepts that are relevant in understanding the behaviour of a software system and influential factors that affect those behaviours.

Software comprehension, which historically has been associated with program comprehension and reverse engineering, involves extracting and representing the structural and behavioural aspects of the implementation in an attempt to recreate the intended design of the software. Software comprehension is motivated by the fact that the software may need to be (1) verified to ensure that the implementation is behaving as it was designed to behave; (2) maintained to fix bugs or make modifications; or (3) redesigned and evolved to improve performance, reusability or extensibility (among other reasons). In order to perform these tasks, an understanding of the current implementation is required and is attained using reverse engineering (RE) tools and techniques.

RE tools (e.g. Rigi [2] and PBS [3]) analyse the implementation at a very low abstraction level (i.e. at the source code level) and, thus, are inappropriate for agent software because they produce models of the implementation that are too detailed (e.g. component dependence and class inheritance models). Besides being limited to supported programming languages, these tools do not provide abstracted views of the implementation as a whole in terms of high-level agent concepts (e.g. beliefs, tasks, goals and communication messages). Wooldridge states that as software systems become more complex, more powerful abstractions and metaphors are needed to explain their operation because "low level explanations become impractical" [4]. To attain an understanding of agent behaviour, the models resulting from the comprehension process must be at the abstraction level at which agent concepts are the elemental or base concepts.

In addition to static analysis of the source code, dynamic analysis tools (e.g. SCED [5] and Hindsight [6]) can create flowcharts, control-flow and state diagrams. However, these tools also face the same problem of detailed representation of programmatic concepts such as process threads, remote procedure calls and data structures, rather than agent-oriented models of goals, plans and interaction protocols. Dynamic analysis is particularly important for agent systems that operate in the presence of environmental dynamics and uncertainty. This research leverages agent concepts to build abstract representations of the agents' run-time behaviour (i.e. relational graphs), which can be quickly understood by the user and can also be used for automated reasoning to further assist the user.

To deal with the large amount of data resulting from source code or execution analysis, some RE tools (e.g. SoftSpec [7]) allow users to query a relational database of gathered data. However, most RE tools leave it up to the user to parse, interpret and digest the data. The research described in this paper deals with the large amount of data by automating data interpretation for the user.

Instead of a list of unconnected, detailed data that the user must relate manually, the presented solution automatically relates run-time observations together in a causal graph. This is similar to the GUPRO toolset [8], where source code is transformed into graphs, except that the graphs nodes are in terms of agent concepts.

As described, RE tools only produce representations of the implementation and have no model of the user's comprehension. It is the user's responsibility to digest the RE results (e.g. diagrams, charts and databases). RE tools do not reflect how much the user understands and, thus, cannot provide feedback to the user about the user's comprehension. However, in model-checking, the user expresses their understanding of the implementation as a "model", which can be automatically checked for specified properties. Thus, model-checking tools have a representation of the user's comprehension of the system. Though useful due to the exhaustive state-space search, model-checking techniques in general do not verify the accuracy of the "model" with respect to the actual system (often referred to as the translation gap problem). Hence, any checked properties may not apply to the actual implementation. Additionally, the model must be made simple enough such that the model-checker can search the entire state-space. By combining model-checking with reverse engineering, this research maintains a model of the user's comprehension (as the user is learning about the implemented agent system) and also ensures that the model accurately represents the actual system.

3 Building the Interpretation

When a user tries to comprehend agent behaviour in the implemented system, the user is essentially building an interpretation by observing and examining agent actions, communicated messages, environmental events and any other run-time data that can be acquired from the implementation. As shown in Fig. 1, background knowledge about the expected behaviour of the implemented system is required to relate the otherwise unconnected observations together. Background knowledge K represents the user's comprehension of the system, which is commonly derived from many sources, such as specifications of the design, experience with the implementation and intuition from presentations. In model-checking, K is a model that is to be checked and it is manually specified by the user.

In this research, K is modelled using a semantic network (i.e. directed graph) of agent concepts that are interconnected by causal relations. The current set of agent concepts includes *goal*, *belief*, *intention*, *action*, *event* and *message* – the set can be extended to include other concepts that may be of interest to the user. For example, in Fig. 1, the background knowledge for an agent's behaviour denotes an intention that is influenced by two different beliefs (denoted by a circle and square). The intention causes an action to occur, which in turn affects one of the beliefs.

This research takes advantage of agent concepts to create interpretations of agent behaviour in the implemented system. Note that K represents a

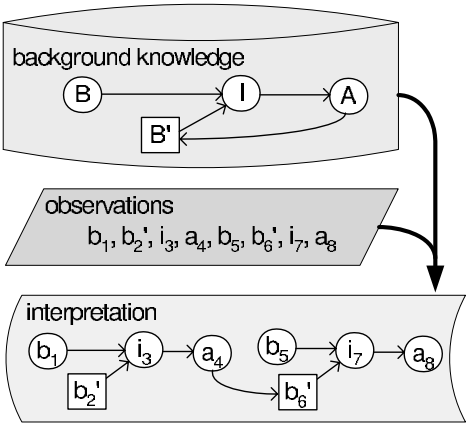

Fig. 1. An interpretation for an agent, given the background knowledge K and observations O_sSS

behavioural pattern and, thus, can have cycles in the graph. However, the interpretation, which consists of actual observations and their relationships, does not have cycles.

To build an interpretation, observations are mapped to agent concepts in K and are linked together using relations defined in K. For example, observations b_1 and b_5 are mapped to agent concept B because the observations are beliefs about a target's state; b'_2 and b'_6 are mapped to B' because the observations are beliefs about the target's location; i_3 and i_7 are mapped to I; etc. Since I is causally related to B and B', directed edges are added between the appropriate nodes (e.g. from b_1 and b_2 to i_3) to relate the observations together. In other words, since the user expects beliefs about a target's state B to influence the agent's intention I, the user will create an interpretation where the corresponding observations for that agent are causally linked.

Background knowledge K is constructed by the user and describes how the agents are expected to behave in terms of the agent concepts. As shown in Fig. 2, the manual procedure for building comprehension can be expressed as

$$K' = update_{manual}(K, D, I, O_s) \tag{1}$$

where K is the previous background knowledge, D denotes the design models and documentation, I is the implementation expressed in source code, and O_s is a set of observations resulting from executing the implementation I in some scenario s:

$$O_s = observe(execute(I, s)) \tag{2}$$

Note that since comprehension is an iterative process, construction of K' involves modifying and updating the previous background knowledge K. To build up comprehension, the user has the tedious task of gathering, organizing and relating the data from the design D, the implementation I and the observations O_s.

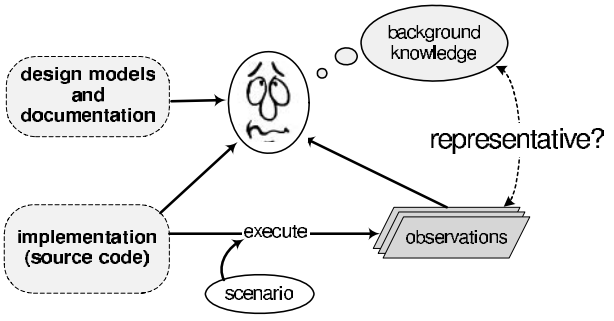

Fig. 2. Manual software comprehension

Due to human error or outdated design specifications, system behaviour described by K may be erroneous or inaccurate with respect to the actual behaviour of the system, particularly as the implementation is updated and maintained over time. To generate accurate interpretations, K must accurately reflect the implementation's actual behaviour. Using empirical techniques, the user must manually verify that the expected behaviour expressed as K is representative of the actual behaviour from the implementation. Due to complexities and uncertainties of some systems, agent behaviours cannot always be predicted from only the design specification in general [9]. Thus, the construction of K must incorporate empirical studies of the implementation.

The overall approach of this research is to build up the background knowledge K using observations from the actual implementation's executions, rather than relying on design specifications as it is in model-checking. As a result, everything in K is based directly on the actual implementation (similar to the RE approach). Modifications to K (e.g. addition of relations between agent concepts) are automatically suggested by the Tracer Tool. However, unlike RE, where detailed models are automatically created for the user to digest, this approach demands that the user confirms all modifications to K so that K also reflects what the user comprehends. In other words, since the user is building K, there is nothing in K that the user does not already comprehend or at least has seen. Consequently, the user does not have to digest all interpretations. Any new or inconsistent behaviours are automatically detected and brought to the attention of the user. Additionally, automatically generated suggestions and explanations can help the user deal with the anomalous behaviour.

The following describes the overall approach taken by this research to ensure the representativeness of the background knowledge. Functions begin with a lowercase letter (e.g. $interpret(K, O_s)$), while predicates begin with an uppercase letter (e.g. $Consistent(K, N_s)$).

As seen in Fig. 3, the reverse engineering approach helps the user by analyzing O_s to produce interpretations N_s, which consists of models derived from observations O_s resulting from actual system behaviour:

$$N_s = interpret_{RE}(O_s) \tag{3}$$

However, the user still has the task of ensuring that K accurately represents N_s.

To aid the user in software comprehension, this research automates the tasks of interpreting the observations *with respect to* K (and in the process, verifying K) and suggesting modifications to K (see Fig. 4). This is possible by explicitly modelling the user's background knowledge K and using it as input to the Tracer Tool. Thus, the new *update* function is

$$K' = update(K, D, N_s, k) \tag{4}$$

where interpretation N_s is derived by mapping the observations O_s to agent concepts in K:

$$N_s = interpret(K, O_s) \tag{5}$$

and the set of suggestions k consists of relations that can be added to the background knowledge K:

$$k = suggest(N_s) \tag{6}$$

Since background knowledge K should accurately model the user's comprehension, the user remains in control of K and must confirm all suggestions before K is modified. However, the user no longer needs to directly analyse the observations O_s from the implementation execution or verify that K accurately reflects the implementation's behaviour, as these tasks are automated by the Tracer Tool. With the interpretations N_s readily available, the user can modify K as they see fit. Through each iteration of building up K, the Tracer Tool verifies K against the observations O_s in case the user introduced errors into K.

If the implementation's behaviour changes (resulting from design changes or maintenance tasks) and is different from the expected behaviour represented by K, the Tracer Tool alerts the user of the new or inconsistent behaviour in N_s and generates suggestions for updating K. Since changes to the implementation can be propagated to K, the accuracy of K with respect to the implementation is maintained as the implementation evolves.

Formally stated, the background knowledge K is representative of the implementation I if and only if K is complete and consistent with respect to interpretations N_s for each execution scenario s in a set of scenarios S:

$$\begin{aligned} &Representative(K, I, S) \iff \\ &\quad \forall s \in S \ (Complete(K, N_s) \wedge Consistent(K, N_s)) \end{aligned} \tag{7}$$

where $Complete(K, N_s)$ is true if there is no suggestions for updating K (i.e. $k = \emptyset$) and $Consistent(K, N_s)$ is true if there are no contradicting behaviours. Ideally, S would be a *complete* set of scenarios covering all possible threads of execution the implementation would encounter. Since this is not usually feasible, a scenario set that covers a reasonable number of execution threads is assumed to be given.

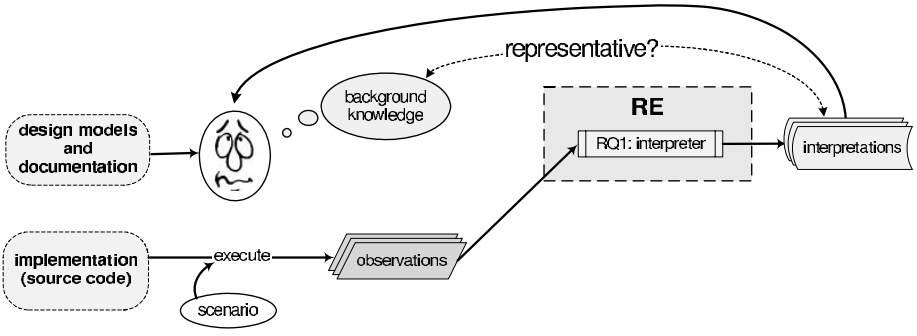

Fig. 3. Reverse engineering approach

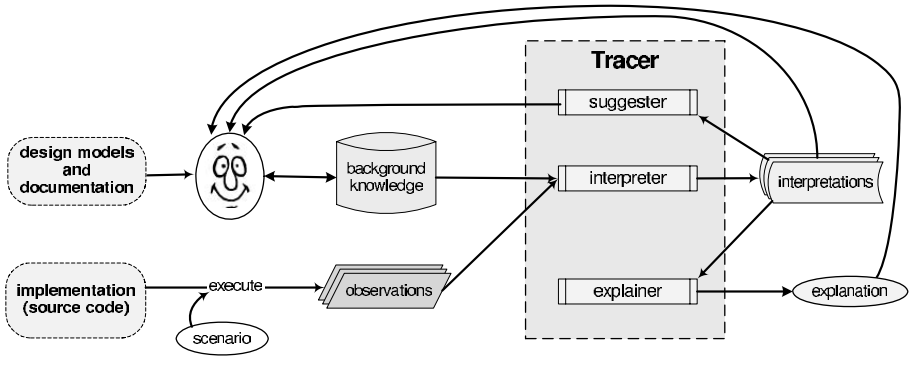

Fig. 4. Automated interpretation approach using Tracer

4 Tracer Tool

To generate accurate interpretations, the background knowledge K should be representative of what is being explained (i.e. agent behaviour in the implementation). This implies that the K (representing expected agent behaviour) must be complete and consistent with the implementation's behaviour (Eq. 7). By explicitly modelling the user's comprehension as K, the accuracy of K can be verified during the interpretation process, which has been mostly automated by the Tracer Tool.

The Tracer Tool addresses the comprehension issues (described in Section 2) in the following ways:

low abstraction level: Background knowledge K is represented as a collection of high-level agent concepts familiar to designers, developers, and end-users.

language-dependent: The Tracing Tool records observations logged from the implementation's execution, rather than analyzing language-dependent stack traces and process threads.

large amount of data: The Tracer Tool automates the task of collecting, organizing and interpreting the observations and can present the interpretation to the user as a relational graph that can be quickly digested.

human user must digest data: Given interpretations and K, automated reasoning can highlight new concepts and relations that the user has not yet modelled in K and ignore already modelled relations.

The following subsections describe the Tracer Tool with respect to Eqs. 2, 5, and 6.

4.1 Equation 2: $O_s = observe(execute(I,s))$

Since K and N_s are modelled at the agent concept abstraction level, only agent concepts are extracted from the implementation – more detailed concepts (e.g. data structures and method calls) are not needed. To acquire only the agent concepts from the implementation, the approach is to instrument the source code (i.e. add extra code to log data). The extra logging code (generated by the Tracer Tool) is inserted at locations where agent concepts occur or change. When the implementation is executed in a scenario s, only agent-relevant data are logged as observations O_s, which are collected by the Tracer Tool. By not parsing the implementation's source code, the Tracer Tool can operate with any software system implemented in practically any mix of languages. This reverse engineering approach requires only a high-level structural understanding of the implementation and encompasses the entire agent system implementation rather than just portions of the code. Scalability is better than reverse engineering because only relevant data about the system are analysed, not every method call or data structure change. This approach translates run-time data and occurrences from the implementation execution into observations of agent concepts. Since the observations are coming from numerous agents and may be out of order, the Tracer Tool sorts and organizes the observations (during run-time) for the next step, which is creating the interpretations.

4.2 Equation 5: $N_s = interpret(K, O_s)$

To produce interpretations N_s from the implementation, observations O_s of the implementation execution are used as shown in Fig. 4. That is, actual observations are mapped to concepts defined in K. If relations exists among the concepts, then concrete relations (shown as directed edges) are defined between the observations. Instead of having the user manually organize and relate observations, the Tracer Tool automatically collects and interprets the observations for the user by linking observations with each other based on the explicitly modelled background knowledge K. If K is initially empty or minimal, the Tracer Tool will suggest updates for K to the user, as described in the next section. In this case, the interpretations are semantic graphs with observations as nodes. Run-time attributes of the the observations, such as observation type and name, time-stamp and belief values, are used to map observations to agent concepts in K. If a relation exists between two agent concepts according to

K, a directed edge is created between the corresponding observations. In [10], a detailed demonstration of creating interpretations using the Tracer Tool is described. In [11], the Tracer Tool is combined with a temporal logic tool to discover behaviour violations in an UAV (Unmanned Aerial Vehicle) domain.

Essentially, K is being used as a template for creating the interpretation N_s of an execution trace. K is a representation of expected behaviour, while interpretation N_s is a representation of actual behaviour. If there are inconsistencies between the N_s and K, then K may need to be modified, similar to the changes that need to be made to the user's comprehension if the implementation does not behave as expected. Suggestions provided by the Tracer Tool can help the users correctly modify K.

4.3 Equation 6: $k = suggest(N_s)$

Since the interpretation process performs the mapping between the observations O_s and agent concepts in K, K is verified against the implementation I. If there exists an observation $o \in O_s$ that cannot be related to some other observation based on defined relations in the current K, then a suggestion is offered by the Tracer Tool to update K so that o is a consequence of some other observation. This happens when there is no incoming relation for the agent concept corresponding to the observation o. Beginning with o, the relations-suggesting algorithm searches temporally backwards through the observation list to determine if a previous observation is related in some way to o using heuristics. The heuristics leverage the typical relationships among agent concepts. For example, if o is an action, then the algorithm searches for the last observed intention i that has some similar attribute as those of action o. If such an intention is found, a relation from i to o is suggested.

If there is no suggestion (i.e. $k = \emptyset$), then K is complete – all actual behaviours are modelled by the expected behaviour representation of the background knowledge. If K is not representative of the implementation's behaviour ($\neg Representative(K, I, S)$) and suggestions do not help, then K and/or the implementation need to be manually modified since neither K nor the implementation is assumed to be correct. For example, if the user expects (as modelled in K) an agent to have a belief called 'target location' before the agent creates an intention involving the associated target, and the belief is not in O_s, then the implementation may need to be updated to ensure that the belief 'target location' is actually being ascertained by the agent. On the other hand, K may need to be updated according to design changes that may have occurred during development that were not incorporated into K. This type of inconsistency is manifested as a missing incoming edge for the intention observation in the semantic network interpretation. However, experiments show that the generated suggestions can correct most of the representative errors in K [12].

Table 1 enumerates completeness and consistency problems between K and N_s that can be identified with the help of Tracer Tool. Causes and solutions for those problems are also listed. Nodes are observations when referring to N_s and agent concepts when referring to K; and edges refer to relations between nodes.

The Tracer Tool offers suggestion for all observations without a (causal) relation from another observation – nodes with no incoming edge, which are detected by the tool. Note that the Tracer tool cannot verify whether *all* causal agent concepts have been identified – such information is application-dependent and relies on the user's knowledge of the domain.

Table 1. Possible completeness and consistency problems between K and N_s

symptom	**cause**	**Tracer's solution**
node in N_s is missing in K	user logged an observation that has no corresponding agent concept in K	Tracer adds the agent concept and suggests relation(s) that link the new agent concept to other agent concepts in K.
edge in N_s is missing in K	not possible since edges are created only if a corresponding relation exists in K	not applicable
node in K is missing in N_s	observation did not occur in the scenario; or user did not correctly insert the corresponding logging code; or user incorrectly added the node in K	not considered an error by Tracer because there is no inconsistency – K models a superset of behaviours exhibited in N_s. The node may appear for an interpretation of another scenario.
edge in K is missing in N_s	the relation did not occur in the scenario	not considered an error by Tracer because there is no inconsistency. The edge may appear for an interpretation of another scenario.

5 Using Interpretations

Explanations of agent actions offer an understanding of why agents behave in a certain way in a given scenario. An explanation of agent behaviour answers a question like "Why did agent action m occur?" A desirable explanation could be "Action m was performed by agent n_1 because n_1 believed belief b, which was due to the occurrence of event e, which was an expected consequence of agent n_1 performing action a, which was planned as a result of negotiations with agent n_2 about n_2's goal g." Other relevant agent concepts can include details about communication messages and updated beliefs resulting from the messages.

Since there is no direct way to measure how much the user comprehends, a person's comprehension of a subject is indirectly measured by how much the person can explain about the subject because the process of creating an accurate explanation demands correct comprehension of the system. Explanations bridge the gap between expected and actual behaviour (i.e. between the explainer's background knowledge and the implementation's execution). Thus, explanations can be very important in designing, debugging, and trusting agent behaviour.

Unfortunately, ensuring accurate explanations is difficult because the implementation evolves over time and there are many factors that can influence

agent behaviour. Firstly, since comprehending the behaviour of the implemented system relies on how accurately the background knowledge represents the implementation, the representative accuracy of the background knowledge must be maintained as the implementation changes. The second problem in manual explanation generation is that an explanation may be too difficult to conceive due to the sophistication (e.g. in reasoning or agent interaction) of the agent system or the amount of observed data to consider. In response to these difficulties, this research proposes an automated approach to agent software comprehension that can handle large amounts of observation data and that can automate the generation of explanations to aid the user in comprehending the system as the implementation evolves over time.

Once background knowledge K has been checked for representative accuracy over the chosen set of scenarios S, K can be used to accurately explain an observation (called the manifestation $m \in O_s$), such as an agent action, that occurred in a specific scenario s using observations O_s (resulting from the scenario in which m occurred). An explanation ϵ consists of a subset of observations from O_s and relations among those observations that contributed to (i.e. caused or influenced) the occurrence of m. The relations are derived from K, which defines relations among agent concepts. Thus, explanation generation involves mapping observations to agent concepts and following the relations (backwards) from m to observations that caused m.

Based on the approach illustrated in Fig. 4, an explanation ϵ for manifestation $m \in O_s$ (e.g. agent action) can be generated using the checked background knowledge K and observations O_s (arrows not shown in Fig. 4). To generate an explanation for an observation m, the explainer uses the same technique as in interpretation – mapping observations to agent concepts in K and using relations in K to link observations together. As shown in Fig. 4, if an interpretation N_s of the scenario exists (which it often does in practice), the same explanation can be generated more quickly using N_s because $interpret(K, O_s)$ has already done the work of mapping and relating the observations. Whereas an interpretation (of an agent) displays all recorded observations (for the particular agent) and the relations between the observations, an explanation shows only the casual factors (other observations) that are relevant to the manifestation being explained. Starting from observation m in the interpretation N_s, the explanation is generated by identifying observations that cause or influence the occurrence of m by following edges pointing to m. Recall that the background knowledge K defines the (causal) relations among observation types (i.e. agent concepts). This can be performed recursively to an arbitrary depth to find causes of causes.

$$\epsilon = explain(m, N_s) = explain(m, K, O_s) \tag{8}$$

From Eq. 8, the accuracy of the generated explanations is dependent on the representative accuracy of the K, specifically on how accurately K reflects the context of what is being explained (i.e. the implementation I) – thus, stressing the need to maintain the representativeness between K and I as described in Section 3. This justifies the value of maintaining an accurate K, in addition to modelling the user's comprehension.

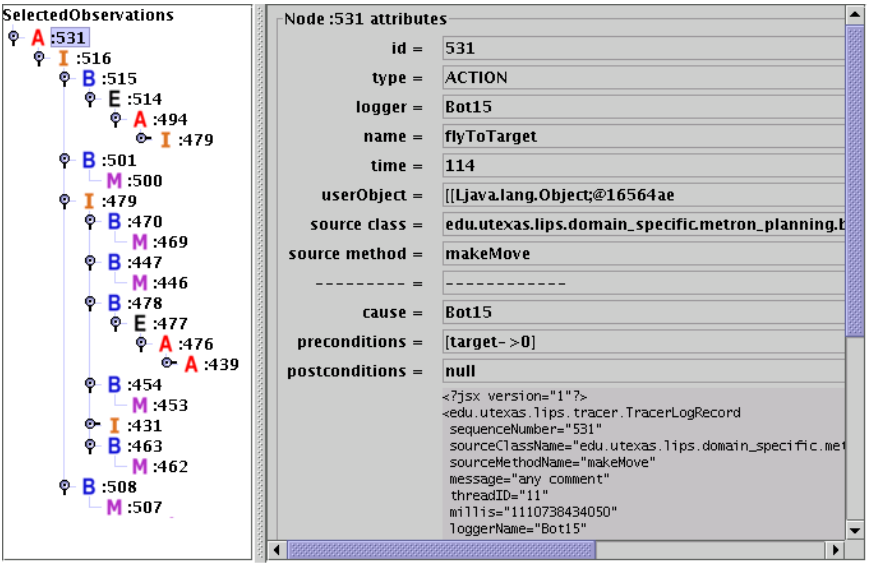

Fig. 5. Explanation in Tracer

Since background knowledge K is represented using agent concepts, the generated explanations will be in terms of the same high-level agent concepts, understandable by anyone with a general knowledge of agents. The explanation can be expressed as a tree graph (as seen in Fig. 5), where the root of the tree is the observation m that is being explained. Child nodes are observations that influenced or caused the parent node observation to occur. The depth of the explanation tree continues until an observation with no causal observation exists, which is one of the initial observations or an exogenous event that independently occurs in the environment. If the explanation does not end with one of these observations, then K may be incomplete and require relations to be added.

Explanations can help focus on and track down the cause of the undesirable behaviour. With explanations readily available to the user, tasks such as redesigning, debugging and understanding agent behaviour becomes a more manageable task and less prone to human error.

6 Summary

The objective of this research is to help users (i.e. designers, developers, and end-users) comprehend agent behaviours within agent-based software systems. This paper describes an approach to automate the process of interpreting agent behaviour. Borrowing the model-checking approach, a model of the user's comprehension (i.e. background knowledge) is maintained as the user is learning about the implemented agent system. Using the reverse engineering approach, the background knowledge is verified against the actual system using observations

of the implementation execution. In this way, the correctness of the background knowledge can be given as feedback to update the user's comprehension of the system.

The contribution of this paper is a practical method to produce a model that (1) accurately represents the actual system (i.e. the implementation) in terms of agent concepts familiar to the designer, developer, and end-user, (2) explicitly represents the user's growing knowledge of the software's behaviour, and (3) can be used for automated reasoning to reduce the effort of software comprehension. The method describes a process to create, refine and verify the user's comprehension of the system with respect to the implementation. With the aid of the Tracer Tool, many of the manual tasks are automated, such as verifying expected behaviour, scanning for unexpected behaviour and generating explanations. With the verified background knowledge, accurate explanations of actual agent behaviour that are consistent with run-time observations can be generated. The Tracer Tool generates interpretations and explanations as evidence of software comprehension and allows the user to analyse reasons for agent behaviour, thereby facilitating software maintenance tasks and promoting confidence in the adoption of agent technology.

Acknowledgments

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

References

1. Stroulia, E., Syst, T.: Dynamic Analysis for Reverse Engineering and Program Understanding. ACM SIGAPP Applied Computing Review **10**(1) (2002) 8–17
2. Agrawal, A., Du, M., McCollum, C., Syst, T., Wong, K., Yu, P., Mller, H.: Rigi - An End-User Programmable Tool for Identifying Reusable Components. In: 5th International Conference on Software Reuse, Victoria, British Columbia (1998)
3. Finnigan, P.J., Holt, R.C., Kalas, I., Kerr, S., Kontogiannis, K., Meller, H.A., Mylopoulos, J., Perelgut, S.G., Stanley, M., Wong, K.: The Software Bookshelf. IBM Systems Journal **36**(4) (1997) 564–593
4. Wooldridge, M.: An Introduction to MultiAgent Systems. John Wiley and Sons, Chichester, England (2002)
5. Koskimies, K., Mnnist, T., Syst, T., Tuomi, J.: Automated Support for Modeling OO Software. IEEE Software **15**(1) (1998) 87–94
6. Hindsight: http://www.testersedge.com/hindsight.htm. (2004)

7. Bruening, D., Devabhaktuni, S., Amarasinghe, S.: Softspec: Software-based Speculative Parallelism. In: 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization, Montery, California, ACM Press (2000)
8. Kullbach, B., Winter, A.: Querying as an Enabling Technology in Software Reengineering. In Nesi, P., Verhoef, C., eds.: 3rd European Conf. on Software Maintenance and Reengineering, Los Alamitos, IEEE Computer Society (1999) 42–50
9. Jennings, N.R.: On Agent-based Software Engineering. Artificial Intelligence **117** (2000) 277–296
10. Lam, D.N., Barber, K.S.: Debugging Agent Behavior in an Implemented Agent System. In Bordini, R.H., Dastani, M., Dix, J., Seghrouchni, A., eds.: Lecture Notes in Computer Science. Volume 3346. Springer-Verlag (2005) 103–125
11. Lam, D.N., Bosse, T, Barber, K.S.: Automated Analysis and Verification of Agent Behavior. In: 5th International Conference on Autonomous Agents and Multiagent Systems, Hakodate, Japan (2006) 1317–1319
12. Lam, D.N., Barber, K.S.: Comprehending Agent Software. In: 4th International Joint Conference on Autonomous Agents and Multi-Agent Systems, Utrecht, Netherlands (2005)

A Semantic and Pragmatic Framework for the Specification of Agent Communication Languages: Motivational Attitudes and Norms

Rodrigo Agerri[1] and Eduardo Alonso[2]

[1] School of Computer Science, University of Birmingham
B15 2TT, Birmingham, UK
r.agerri@cs.bham.ac.uk
[2] Dept. of Computing, City University
EC1V 0HB London, UK

Abstract. The ability to communicate is one of the most important properties of agents. In an open environment, like the Internet, in which agents are designed in many different ways, it is important to clearly establish the meaning of a standard language for artificial agents. Traditionally, the pragmatics of ACLs take the form of interaction protocols, which only specify the order in which messages occur without taking into account the content of the message, or the role of the agents. We present a unified ACL which attempts to define the ACL semantics and pragmatics within the same framework, including an intentional view of speaker's meaning and a pragmatic level based on the normative notion of right. The framework is developed by defining a logic with modal and deontic operators grounded in a computational model. The pragmatics takes the form of declarative rules.

1 Introduction

The adoption of a standard Agent Communication Language (ACL) is crucial for artificial agents to interact in open environments. Communication is a kind of interaction that should not affect the autonomy or heterogeneity of the agents. This is particularly true in open environments, such as electronic commerce applications based on the Internet, where agents are designed by different constructors and work for their individual interests.

Most of the approaches to ACLs are based on speech act theory [1]. According to this theory, linguistic communication is just a special type of action that can be seen from three different points of view. An *illocution* is the central component of a communicative action and it corresponds to what the action is intended to achieve. This goal should be distinguished from the effect that the communicative action is meant to produce on the receiver (*perlocution*), as well as from how the actual communication is physically carried out (*locution*). In this paper, we aim to introduce a framework in that the semantics consists of a complete catalogue of communicative actions with a complementary pragmatic component that accounts for the social effects of performing a communicative action and thereby facilitates the achievement of its perlocutionary effects.

Current approaches to ACLs can be classified according to three different views: KQML (Knowledge Query Manipulation Language [2]) and FIPA ACL (Foundation

M. Kolp et al. (Eds.): AOIS 2005, LNAI 3529, pp. 16–31, 2006.

for Intelligent Physical Agents [3]) are based on a *mentalistic* approach. The meaning of the communicative actions is defined in terms of the private mental states of the agents. The semantics of the cognitive operators for beliefs, intentions etc. are given using possible world semantics, which means that the resultant ACL semantics are not public. Furthermore, it has been argued that, in open environments, in which agents are heterogeneous and competitive, it is not sensible for agents to trust their opponents in a negotiation process by making assumptions about their current beliefs or intentions [4,5]. The second approach, known as *procedural*, focusses on the design of conversation templates. ACLs are defined in terms of message sequences, that is, the meaning of a communicative actions depends on the order in which can be used. Examples of this approach can be found in [6] and [7], among others. It has been claimed that procedural accounts over-constrain agents' behaviour, transforming communication in a meaningless exchange of ordered tokens [4]. Finally, the *social* approach takes into account the social consequences derived from performing a communicative action. For instance, the commitments that agents acquire by sending a particular message. Some authors take *commitment* as the core social notion to define the meaning of the speech acts [4,5]. The general idea is that by defining the ACL in terms of commitments, the ACL is made public.

The social approach is a very promising approach to the specification of public ACLs but we also believe that some of the criticisms of the mentalistic approaches are the result of a misconception, namely, that the ACL semantics should also capture the social character of communication, or that it should achieve the perlocutionary effects. Traditionally, mentalistic and social approaches are semantic based. Sometimes some interaction protocols are also provided (Contract-Net, English Auction etc.) to specify the order in which speech acts are to be performed. Procedural approaches are purely protocol-based, so the resultant protocol is not high-level enough for the ACL to be used in open multi-agent systems. In our view, the ACL pragmatics should complement the ACL semantics not only by specifying conversation templates, but also by contextually regulating the use of the speech acts in a way that it provides a method for the achievement of the perlocutionary effects.

A number desiderata for ACLs have been proposed by various authors. Table 1 summarizes the most common requirements that ACL should satisfy and it compares some ACLs proposed so far.

Although the social approaches perform well with respect to some of the requirements, they still do not provide a pragmatic theory to regulate the use of the speech acts in conversation. None of these approaches guarantee the fulfilment of the perlocutionary effects. Dealing with autonomous agents, it is not possible to guarantee that the perlocutionary action is satisfied, because its fulfilment depends on the receiving agent.

The remainder of the paper is structured as follows: In the next section, we introduce the main concepts of a unified ACL framework. In Section 3 a formal definition of the semantics is given. Section 4 defines the pragmatic specification language and proposes the use of conversation policies and interaction protocols within the framework provided. Section 5 discusses how our proposal compares to related approaches and some conclusions and further work are discussed.

Table 1. Desiderata for ACLs

Requirements	**ACLs**		
	FIPA ACL	Procedural Singh (2000)	Fornara and Colombetti (2004)
Autonomous	-	✓	✓
Complete	-	✓	✓
Contextual	-	-	-
Declarative	✓	✓	-
Formal	✓	✓	✓
Grounded	-	?	✓
Public	-	✓	✓
Perlocutionary	-	-	-

2 General Framework

Agent Communication consists of agents exchanging messages which are well-formed formulae of a communication language $\mathcal{L}_c$. By sending messages, agents perform speech acts. Speech Acts are usually classified in terms of their illocutionary point [1].

A well-defined semantics is a central component of the specification of an ACL. Traditionally, ACLs consist of a set of communicative actions and several interaction protocols that define conversational templates for specific scenarios (e.g. auctions). This means that it was generally assumed that the meaning of speech acts could be semantically defined for every context. Social approaches aim to provide public ACLs, but the absence of a complementary pragmatic component to the semantics means that some issues such as the achievement of the perlocutionary effects are still not solved. We claim that the ACL pragmatics should not only consist of the interaction protocols but it should also define policies that contextually enrich the minimal meaning of the speech acts. As Singh [4] put it

> "What we usually refer to informally as *meaning* is a combination of the semantics and the pragmatics. We will treat the semantics as the part of the meaning that is relatively fixed and minimal. Pragmatics is the component of meaning that is context-sensitive and depends on both the application and the social structure within which is applied. [...] Pragmatic claims would be based on considerations such as the Gricean maxims of manner, quality and quantity."

We believe that a declarative ACL pragmatics with these characteristics is needed. Our unified ACL consists of a set of speech acts, the Speech Acts Library (SAL), and the ACL pragmatics, which are structured as a set of conversation policies and interaction protocols. The ACL pragmatics are called NPRAG (from Normative PRAGmatics) and it consists of normative rules to regulate agents' behaviour. The key concept of NPRAG is a notion of right defined specially for our tasks. We also define two specification languages, $MCTL_I$ and $NCTL_I$, to define the semantics of the cognitive and normative concepts used in the ACL. The ACL semantics encodes the intentional character of communication between autonomous agents. The ACL pragmatics contextually regulates the use of the speech to facilitate the achievement of the perlocutionary effects. Thus, a unified ACL is defined as the tuple

$$\mathcal{UACL} = \langle SAL, MCTL_I, NPRAG, NCTL_I \rangle$$

Messages of SAL are based on a STRIPS-like language with preconditions and effects. On the one hand, the preconditions are to be true for the agent to send a message (including the goal the sender intends to achieve by sending that message). On the other hand, the effects state the response that the sender wants to produce in the audience. This is a problematic issue because, as has already been discussed, autonomous agents, by definition, cannot be forced to guarantee the effects. The semantics of SAL are given by a function

$$[\![-]\!]_{SAL} : \mathit{wff}(SAL) \rightarrow \mathit{wff}(MCTL_I)$$

The syntax of the communication language SAL is based on the FIPA ACL [3]. The semantics of the motivational and temporal operators will be given by $MCTL_I$ in the next section. The language $MCTL_I$ is based on Computational Tree Logic (CTL) [8] extended with operators for beliefs, goals and intentions. We combine the cognitive notions with the temporal operators à la Fagin *et al.* [9]. In doing so, we are effectively grounding $MCTL_I$ to a computational model, which is the first stage to facilitate its verification [10].

In the interpretation for beliefs, goals and intentions proposed here, these attitudes are ascribed to the agents by an external reasoner about the system. In this approach, agents do not compute their beliefs, goals and intentions in any way and, as a consequence, the ACL defined using $MCTL_I$ as the semantic specification language does not rely on agents' internal (mental) states.

$NCTL_I$ consists of branching temporal operators combined with a deontic for obligations. $NCTL_I$ provides the semantics for the normative operators used in the specification of NPRAG. The semantics of NPRAG are given like the semantics of SAL above. The conversation policies and interaction protocols of NPRAG can be specified using a logic-based declarative language.

In the following sections, we present the ACL semantics. Firstly, $MCTL_I$ is defined showing to the semantics of beliefs, goals and intentions are grounded in a computational model. $MCTL_I$ is then used to define a complete set of speech acts.

3 $MCTL_I$

3.1 Syntax

Definition 1 ($MCTL_I$ **Syntax).** *The BNF definition of* $MCTL_I$ *formulae is as follows. Consider n agents.*

$$\phi := AP | \neg\varphi | \phi \wedge \psi | B_i\phi | G_i\phi | I_i\phi | EX\phi | AX\phi | A[\phi U \psi] | E[\phi U \psi]$$

The boolean operators are standard. E and A are quantifiers over paths, meaning "there exists a run" and "for all runs" respectively. As in CTL, $MCTL_I$ temporal connectives consist of a pair of symbols. The first pair is one of the quantifiers, whereas the second one is F, G, U or X meaning "some future state", "all future states (globally)", "until" and "next state", respectively. Until is a special operator which means that ψ does eventually hold and that ϕ will hold everywhere until ψ holds. As usual, we define F and

G as abbreviations $AF\phi \equiv A[True\ U\ \phi]$, $EF\phi \equiv E[True\ U\ \phi]$, $AG\phi \equiv \neg EF\neg\phi$, $EG\phi \equiv \neg AF\neg\phi$. $B_i\phi$, $G_i\phi$ and $I_i\phi$ are operators for beliefs, goals and intentions respectively.

There are two main semantic approaches to the formalization of agent systems via modal logics. The traditional model is based on the work of [11] on possible-world semantics. It has been argued that possible-world models cannot be related to a computational model The possible-world approach includes the theory of intention [12] and the BDI logic of [13]. Appropriate grounded semantics ensures that a clear correspondence can be found between states in the computing system and configurations in the logical description (see [10] for a good discussion on these issues).

The second approach, the Interpreted Systems model, offers a natural interpretation of the notion of knowledge in terms of states of agents in distributed systems [9]. We adapt the interpreted systems to our purposes of giving a grounded semantics for beliefs, goals and intentions.

3.2 Semantics

The key idea is that agents in multi-agent system are in some state at any point in time. This state is the agent's local state, which consists of all the information about other agents and about the environment to which agents have access (we follow [9] in the definition of Interpreted System).

Furthermore, we can also think of the whole system as being in some state. In this sense, the notion of *environment* refers to everything else in the system that is not an agent. Both the agent's local state and the environment's state conform to the global state of a system.

Definition 2 (Global States). *A tuple $(s_e, s_1, \ldots, s_n)$ represents a global state in a multi-agent system where s_e is the environment's state and s_i is agent i's local state, for $i = 1, \ldots, n$.*

A system evolves over time. Thus, we define a run as a function from time to global states, which gives a complete description of what happens over time in one possible execution of the system. Following this, we define a system as a set of runs. The system is always at a global state at some point.

Definition 3 (Points). *A point is a pair (r, m) consisting of a run r and a time m. We assume that time is discrete and ranges over the natural numbers. At a point (r, m) the system is in some global state $r(m)$. If $r(m) = (s_e, s_1, \ldots, s_n)$, then $r_i(m)$ is the agent's local state at the point (r, m).*

A system can be seen as a Kripke structure except that we do not have any labelling or interpretation function to assign truth values to the atomic propositions.

Definition 4 (Interpreted System). *A system T over GS is a set of runs over global states in GS. A point (r, m) is a point in system T if the run r is in the system T such that $r \in T$. A round m in run r takes place between $m - 1$ and time m. Thus, an interpreted system IS is a tuple (T, L) where T is a system of runs and L is a labelling function which assigns truth values to the atomic propositions at the global states.*

We extend the interpreted system models with beliefs, goals and intentions. Beliefs are given a standard $KD45$ axiomatization relative to each agent. For goals and intentions, we assume a minimal KD axiomatization to ensure consistency.

Definition 5 ($MCTL_I$ Structure). *Given a system of runs T, the structure $MCTL_I$ is generated by associating the interpreted system $IS = (T, L)$ with the serial, transitive and euclidean Kripke structure $M = (S, \mathcal{B}_i, \mathcal{G}_i, \mathcal{I}_i, L)$, such that $MCTL_I = (GS, \mathcal{B}_i, \mathcal{G}_i, \mathcal{I}_i, L)$ where GS corresponds to the sets of global states in IS. L is a labelling function $L : S \rightarrow 2^{AP}$ from global states to truth values, where AP is a set of atomic propositions. $\mathcal{B}_i$ where $i = (1, \ldots, n)$ is a set of agents, gives the accessibility relation on global states, which is serial, transitive and euclidean. Thus, we have that $(l_e, l_1, \ldots, l_n)\ \mathcal{B}_i\ (l'_e, l'_1, \ldots, l'_n)$ if $l'_i \in GS_i$. The serial relations for goals $\mathcal{G}_i$ and intentions $\mathcal{I}_i$ are defined in the same manner.*

We say that a formula ϕ is true at a point (r, m) in a system IS, $(IS, r, m) \models \phi$, iff ϕ is true in the $MCTL_I$ structure generated by IS and M, that is, $(MCTL_I, (r, m)) \models \phi$.

Before we define the semantics of $MCTL_I$, we need an extra definition:

Definition 6 (Extension). *Given an interpreted system IS, we say that $r' \in T$ extends the point $(r, m) \in T$ if both runs r and r' go through the same sequence of global states up to time m. Formally, if $r'(m') = r(m')$ for all $m' \leq m$.*

Definition 7 ($MCTL_I$ Semantics). *The semantics of $MCTL_I$ are inductively defined as follows:*

$(IS, r, m) \models \phi$ *iff* $L(r, m)(\phi) = true$
$(IS, r, m) \models \phi \wedge \psi$ *iff* $(IS, r, m) \models \phi$ *and* $(IS, r, m) \models \psi$
$(IS, r, m) \models \neg\phi$ *iff it is not the case that* $(IS, r, m) \models \phi$
$(IS, r, m) \models EX\phi$ *iff for some run* r' *that extends* (r, m) *then* $(IS, r', m+1) \models \phi$
$(IS, r, m) \models AX\phi$ *iff for all runs* r' *extending* (r, m) *then* $(IS, r', m+1) \models \phi$
$(IS, r, m) \models B_i\phi$ *iff* $\forall(r', m')$ *such that* $(r, m)\ \mathcal{B}_i\ (r', m')$, *then* $(IS, r', m') \models \phi$
$(IS, r, m) \models G_i(\phi)$ *iff for all* (r', m') *such that* $(r, m)\ \mathcal{G}_i\ (r', m')$, *then* $(IS, r', m') \models \phi$
$(IS, r, m) \models I_i(\phi)$ *iff for all* (r', m') *such that* $(r, m)\ \mathcal{I}_i\ (r'm')$, *then* $(IS, r'm') \models \phi$
$(IS, r, m) \models E[\phi U \psi]$ *iff there is a run* r' *extending* (r, m) *such that* $\phi\ U\ \psi$, *that is, there is some* $m' \geq m$ *along the run such that* $(IS, r', m') \models \psi$ *and for each* $m \leq m'' < m'$ *we have* $(IS, r', m'') \models \phi$.
$(IS, r, m) \models A[\phi U \psi]$ *iff for all runs* r' *extending* (r, m), *that run satisfies* $\phi\ U\ \psi$, *that is, there is some* $m' \geq m$ *along the run such that* $(IS, r', m') \models \psi$ *and for each* $m \leq m'' < m'$ *we have* $(IS, r', m'') \models \phi$.

In this framework, "agent i believes ϕ" means that, "as far as agent beliefs are concerned, the system could be at a point in which ϕ holds". In other words, beliefs refer to the runs of the system. The notion of belief used in this paper does not require that the belief be true. Therefore, an agent holding a belief does not automatically made true

the content of the belief. This property is central for an open MAS, where agents have available incomplete and modifiable information.

An "agent i has the goal of bringing about ϕ" means that, "with respect to the agent's goals, the system could be at a point where ϕ holds". Goals can be seen as facts ϕ at a global state that an agent wants to bring about. "An agent i having the intention to bring about ϕ", means that, from the point of view of the agents' intentions, there is a run in which i intends, along that run, to bring about ϕ.

The states accessible through the relation $\mathcal{G}_i$ are a subset of those accessible through the accessibility relation for beliefs $\mathcal{B}_i$. It is quite common that the set of goal states is a subset of those believed possible, such that $G_i \subseteq B_i$. This responds to the common sense claim that there are facts along runs that the agent does not want to bring about. $MCTL_I$ can be now used to define a catalogue of speech acts.

4 Speech Acts Library

Following Searle's taxonomy [1], we classify speech acts into assertives, commissives, directives, declarations and expressives. The last category is not relevant for the purposes of this paper, so it will not be included. Table 2 presents a speech act for each of the other four categories, plus two more (*agree* and *refuse*) that will be used later. The two speech acts at the top, *inform* and *request*, are classified as assertives and directives respectively. *Declare* is a declarative and *promise* is a commissive.

Table 2. A complete set of speech acts

$\langle i, inform(j, \phi)\rangle$ $FP: B_i(\phi) \wedge G_i(B_j(\phi))$ $RE: B_j\phi$	$\langle i, request(j, \phi)\rangle$ $FP: G_i(I_j(EF\phi))$ $RE: EF\phi$
$\langle i, agree(j, \phi)\rangle$ $\langle i, inform(j, I_iE(\phi\, U\, \psi))\rangle$ $FP: I_i\phi\, U\, \psi$ $RE: B_j(I_iE(\phi\, U\, \psi))$	$\langle i, refuse(j, \phi)\rangle$ $\langle i, inform(j, \neg I_iE(\psi\, U\, \psi))\rangle$ $FP: \neg I_iE(\phi\, U\, \psi)$ $RE: B_j(\neg I_iE(\phi\, U\, \psi))$
$\langle i, promise(j, \phi)\rangle$ $FP: I_iEF\phi$ $RE: EF\phi$	$\langle i, declare(j, \phi)\rangle$ $FP: G_i(EX\phi)$ $RE: EX\phi$

We analyse *inform*, *request* and *agree* in detail and we compare them to other formalizations [12,3,4,5].

4.1 Inform

In our approach, agent i informs j that, according to its beliefs, a fact ϕ holds at a point in the system. The preconditions for i to send an *inform* require i to believe that ϕ holds and to have the goal of j believing that ϕ holds. The Rational Effects are that agent j believes that ϕ is true. The main intuition in this analysis is that when one sends an *inform*, one usually believes the content of the speech act and one has the

communicative goal that the receiver comes to believe the content of the *inform* speech act. The Rational Effect (perlocution) state that the response one wants to produce in the audience is that the receiver adopts the belief in ϕ.

We explicitly define the semantics of the cognitive operators upon a computational model. It is well known that a common problem of possible-world semantics is the difficulty of relating them to a system. However, the ACL semantics are based on mental states (e.g, [3,12]) with semantics defined in terms of possible-world models. Furthermore, these approaches use complex languages with dynamic and cognitive operators that greatly increases the complexity of the logic and of the speech act itself. For example, the FIPA specification requires in the preconditions that the sender i is believes that the receiver is *uncertain* about the content of the *inform* message, but it does not require that i has the goal of agent j (the receiver) believing the content of the *inform* speech act. We have not study the axiomatics of $MCTL_I$ for space constrains, but it is well-known that both components (the cognitive and the branching time operators) can be given a sound and complete axiomatization [13].

With respect to the social-based approaches, Singh [4] proposes that an *inform* means that objectively, "the sender commits that the content is true" and, practically, "the sender commits that it has a reason to know the content". Singh's aim is to use commitments to make the ACL semantics public, but in doing so the idea that the sender has the goal that the receiver adopts a belief is missing. This is what we mean by saying that social-based approaches do not capture the goal-based character of communication. Another way of saying this is that the illocutionary aspect of the speech act which we defined as "what the speech act is *intended* to achieve" is lost. The analysis proposed by [5] follows similar lines to Singh, but the semantics of speech acts are no longer declarative but are given operationally. Finally, Singh defines a language that consists of CTL with modal operators for beliefs, commiments and intentions. However, the operators for beliefs and intentions are not grounded in a computational model.

4.2 Request

The above definition of *request* states that when an agent i requests that agent j brings about some proposition ϕ the preconditions to be satisfied consist of agent i having the goal that agent j intends along a run that ϕ eventually holds. The rational effect is that ϕ eventually holds along a run.

We have already made the point about the complexity of the mentalistic formalizations so we will focus on the social-based proposals: Singh [4] defines *request* to objectively mean that "the sender commits that the receiver will commit to making it true" and practicaly that "the sender commits that the receiver has committed to accepting a request from him". Giving this meaning to a request seems a bit *odd*. Since it is not clear anymore why a sender sends a *request*. In our view, the reason is that the sender intends to achieve a communicative goal that will be satisfied if the receiver agrees to perform whatever was requested. Obviously, this cannot be expressed without using cognitive operators. In this sense, the use of precommitments [5] to analyse *requests* fails, in our view, to express that the sender explicitly expresses its interest of having the receiver executing a particular action. In this approach, a *request* is the execution of a public method that creates an empty slot that has to be filled in.

The definition of *agree* is a good example to appreciate the benefits of using temporal logic as specification language. The informal meaning of *agree* according to FIPA is that an agent agrees to do some action when some preconditions hold. Since we aim to define our library of Speech Acts in the spirit of FIPA CAL specification, we use the until operator to express that agent i agrees to bring about ϕ as long as some precondition ψ holds. This form is far simpler than other formalizations of this speech act proposed to date (see [3]).

Following this point, when an agent i promises to agent i that ϕ will be true, this means that agent i intends along a run that ϕ will eventually hold. The perlocutionary effect state that ϕ eventually holds indeed. Finally, when an agent performs a *declare* that ϕ, the preconditions of the act require that i intends along a run that ϕ holds in the next immediate global state. The perlocutionary effect to be achieved is that ϕ holds at the next state.

Note that the ACL semantics proposed has solved some of the problems summarized in Table 1. The crucial point is that $MCTL_I$ offers a grounded semantics to beliefs, goals and intentions (these notions are external).

The rest of the requirements state that the semantics provided by SAL respects the *autonomy* of agents, it defines a *complete* set of speech acts, it does not take into account different *contexts* in which the speech acts are used, it provides a *declarative* and *formal* meaning, and it does not account for the achievement of the perlocutionary effects. The requirements left are that the ACL be public and contextual. This accounts for fact that the perlocutionary effects are not met by the ACL semantics proposed in this section. In fact, we claim that these problems should not be tackled by the ACL semantics but by an ACL pragmatics that complements the ACL semantics by regulating the use of the speech acts depending on the context, scenario, agents' roles etc. Note that this idea allows us to capture the intentional aspect (illocutionary) of the speech acts in the semantics leaving the public and contextual (or social) issues for the pragmatics.

5 Normative Pragmatics

The ACL pragmatic theory proposed in this paper is normative. So, it is only natural to ask: What are the benefits of using norms in ACL pragmatics? We hope that the following example will illustrate the problems of a semantic-based approach to agent communication. Our specification of *inform* given in the previous section states that when an agent i performs this act:

i It believes its propositional content ϕ and
ii It has the goal that the receiver j will eventually come to believe that ϕ holds.
iii The perlocution is that j comes eventually to believe that ϕ holds.

The process of intention recognition would presumably be described as follows: When agent j receives the message, it will assume that the first two preconditions hold. As a consequence, j should believe that i believes that ϕ, if j trusts the sender's message, j will believe ϕ, which corresponds to communicative goal i wanted to achieve. Assuming that agents do this process is, however, too idealistic. Moreover, it is computationally expensive to let agents do all this reasoning.

This is where interaction protocols come to work. Interaction protocols define the sequences in which speech acts can be performed, so that agents can follow the conversational template without doing all this complex reasoning pictured above. Although interaction protocols are necessary for agent communication, most of them restrict agent conversation to a "follow-the-rule" activity, in which a conversation simply becomes in an exchange of meaningless tokens. These approaches are not concerned with the use of speech acts according to their content in specific contexts. This is due to the fact that none of these approaches consider both aspects, semantic and pragmatic, as the two sides of communicative meaning. The basic intuition behind the normative pragmatics presented in this paper is that the intention recognition process described above is to be regulated by means of rights, obligations and permissions. In fact, when we were talking about the process itself we were saying that agents *should* believe this and *should* do that. If we can make policies to take into account contextual information to regulate the use of the semantics, we do not need agents to do all that complex reasoning.

A second question easily arises: Why using rights and not just obligations and/or permissions? The main reason is because we want to give agents enough freedom, but also limit their behaviour. We believe that there is a middle ground between traditional obligations and permissions as defined in standard deontic logic, and that the concept of *right* that we define below is appropriate to capture this. We do not follow any definition of right in the literature, nor we try to provide a solution for any possible ambiguity that could be found in the notion of right, that is, in the fact that right has been usually used to refer to various things. A concept of right will provide a normative notion that helps to coordinate agent communication but that does not completely pre-determine their behaviour. This concept is in some sense close to what [14] calls *strong permission*. An interesting point in the etymological meaning of the word 'right' comes from that *that is fair* or *just*. This sense allows us to talk about a society that is "rightly ordered", for example. When applied to individuals, rights entitle their holders to some *freedom*.

5.1 $NCTL_I$

The pragmatic specification language $NCTL_I$ inherits most of the formal machinery used in the definition of $MCTL_I$. The only difference is that instead of beliefs, goals and intentions, we simply add a deontic operator for obligations within an organizational structure in which agents are assigned a role [15]. Thus, special propositions i rr j, $g_i\ rn_i$ are introduced to mean that agents i and j are role-related by rr, i is a member of group g, and i plays the role rn, respectively. A role is a set of constraints that should be satisfied when an agent plays the role. For example, the role of auctioneer constrains the goals, obligations, permissions and rights of the agent that plays that role. The scope of the role depends on the institutional reality in which it is defined (e.g. auction). A group is a set of agents (roles) that share a specific feature (i.e. being auctioneers). Finally, role relations constrain the relations between roles (e.g. the auctioneer-bidder relation).

Given a finite set of agents Ag, a finite set of group names GN, a finite set RN of role names, a finite set RR of role relations, and a countable set AP of primitive propositions, the syntax of $NCTL_I$ is given by the following BNF expression:

Definition 8 ($NCTL_I$ Syntax)

$$\varphi := AP|\neg\phi|\phi \wedge \psi|O_i\varphi|EX\phi|AX\phi|E[\phi\ U\ \psi]|A[\phi\ U\ \psi]$$

As usual, permissions are defined as the dual of obligations. The axiomatization of obligation is given by the system KD. $NCTL_I$ structures are the result of associating the interpreted system IS with a Kripke structure in which the accessibility relation for obligation is serial.

Definition 9 ($NCTL_I$ Semantics). *The temporal operators of $NCTL_I$ are the same as in $MCTL_I$, so we give only the definition for the new operator.*

$$(IS, r, m) \models O_i\phi \textit{ iff } \forall(r', m') \textit{ such that } (r, m)\ \mathcal{O}_i\ (r', m'), \textit{ then } (IS, r', m') \models \phi$$

The traditional reading of the obligation operator has been something like "agent i is obliged to bring about ϕ". A more recent interpretation is proposed by [16] where an operator $O_i\phi$ expresses the idea that "if agent i is functioning correctly, then ϕ holds", where ϕ can refer to global or local states in the system. Lomuscio and Sergot introduce the Deontic Interpreted System models, where a deontic notion is grounded in an interpreted system. However, they do not include a temporal component in their logic. Given that we have introduced time in our models, we modify slightly their definition and say "the system is at a point in which ϕ holds if agent i acts correctly".

It remains to define the normative notions of $NCTL_I$ that are not primitive. Specifically, we define what it means for some ϕ to be a violation, for an agent i to have the right to bring about ϕ, and an intuitive notion of sanction. In order to define violation, we extend the language of $NCTL_I$ to include the propositional constant V as an abbreviation of the formula defined below. The meaning of the expression $V\phi$ states that ϕ holding in the system at some point is a violation.

Definition 10 (Violation). *From each literal built from a variable ϕ, $V\neg\phi$ means that $\neg\phi$ is a violation at some point (r, m) in the system for some $ns \in NS$, such that NS is a set of norms, iff*

$$O_iE(\phi\ U\ \psi) \rightarrow \neg E(\phi\ U\ \psi)$$

If the system is at a point in which ϕ holds if agent i acts correctly until ψ holds, then $\neg\phi$ holds until ψ holds. Agent i not working correctly means that ϕ does not hold and that constitutes a violation. We can imagine a context in which if an agent i is functioning correctly then it will send an *accept* message to a *request* when some agreement preconditions hold, then agent i does not *accept* the request. In this situation, we say that agent's i not bringing about ϕ violates the pragmatic specification of accepting the *request*. In some cases, agents have their behaviour specified in a way that performing some action does not constitute a violation. Rights give agents some freedom to act in some specific way. In this sense, *rights* are considered here exceptions to obligations. An agent has the right bring about ϕ under some condition ψ if bringing about ϕ is not a a violation ($\neg V(\phi)$). From an external point of view, we say that "there is a point in the system where agent i is functioning rightly if the holding of ϕ does not constitute a violation". We formalize this concept as follows:

Definition 11 (Right). *Let NS be a set of norms $(ns_1, \ldots, ns_n)$, and let the variables of agent Ag contain a set of violation variables $V = V(\phi)$ such that $\phi \in AP$. Agent i's functioning is right when ϕ holds, $R_i\phi$, for some $ns \in NS$ at some global state $r(m)$, $r(m) \in GS$ iff*

$$E(\neg V\phi \, U\psi)$$

Therefore, having the right to bring about ϕ under some precondition ψ means that until ψ holds along a run, then ϕ not being a violation also holds along that run. So, what happens when an agent not functioning correctly or rightly brings about some ϕ, which constitutes a violation? The specific nature of the sanction varies from system to system, and within the same system, from one scenario to another. The general pattern, however, is that the sanctioned agent will have the obligation to do something as a punishment for its violation. For example, agent i wants to participate in a bidding process to buy a property on behalf of some estate agents. Say that to enter the auction, you need to pay some deposit of 1,000 in advance. If the agent (its role is bidder, $bidder \in RN$) wins the auction with an agreed price of 200,000 for the property, but decides to break the agreement and not buying the house after winning the auction, then it agent has the obligation to pay a fine. In this case, the fine can be the 1,000 deposit paid to enter the auction in the first place. The agent with the right to impose fines in this scenario can be the agent playing the role of auctioneer $auctioneer \in RN$. We can formalize this notion of sanction as follows:

Definition 12 (Sanction). *Let b denote the role of bidder such that $b \in RN$, then a agent i such that $i \in Ag$ playing the role of bidder b has the obligation to pay a fine (by bringing about ϕ) iff*

$$b_i \wedge (O_i E(\phi \; U \; \psi) \wedge \neg(E\phi \; U \; \psi)) \rightarrow O_i\omega$$

Once the notions of obligation, right, permission etc., are defined by NCTL, the next step is to use them to specify a set of normative interaction protocols and conversation policies (NPRAG) for agent communication to complement the Speech Acts Library of the previous section. Note that the main aim of this paper is to define the general framework in which meaning is defined by the combination of the ACL semantics and pragmatics and where the semantics of the two specification languages are grounded in a computational model. Due to space constraints, the next section briefly outlines the development of interaction protocols and policies within the framework provided.

5.2 Policies and Protocols

In our approach, interaction protocols are seen as conversational constitutive rules that specify the structure of conversations. Conversation policies make coordination easier by assigning rights and obligations on the participants, and specifying which speech acts are appropriate at certain points (the agent playing the role of auctioneer establishes the rights and permissions of the participants). The conformity of the participants to the protocol is based on the content of the speech acts used. Conversation policies specify the communicative actions that agents should perform when functioning correctly. Thus, these rules includes contextual information, they make conversations public by

means of social rules and they state that, when agents act correctly, some course of action should be taken. Among these, those actions that satisfy the perlocutionary effects of the speech acts.

We propose the use of a logic-based declarative language (such as Prolog) to specify the protocols and policies. We could have thought of using $NCTL_I$ to define rules that could be capture by finite-state automata. However, it is very difficult to do so when we are talking of assigning roles, rights etc. Prolog offers an intuitive method of expressing rules to build automata, which could prove useful for issues of verification.

Conversation policies involved agents, roles, speech acts, contextual-dependent actions (e.g. bidding), transition actions (for turn-taking), conflict resolution actions (to override conflicts between rules) and normative rules. Contextual actions are expressed in terms of goals, preconditions and effects:

$$done(Agent, Goal, Precondition, Effect)$$

Contextual actions are those actions specific of the institution (e.g. bidding), in which the interaction is taking place, that we call context-dependent actions. Broadcasting actions depend on the platform in which agents run. That is, broadcasting actions are defined by the programming language in which agents are built. For example, in Java built platforms like JADE, sending messages is simply a case of creating an ACLMessage, setting the parameters (sender, receiver, reply-to, performative, etc.) and then sending it using the send() method in the agent object.

Finally, normative rules consist of a deontic operator (obligations, rights) and a condition that has to be true for the rule to be applicable:

$$right(X, request(X, Y, Condition))$$

Agents hold the right to bring about ϕ as long as it does not constitute a violation. An obligation rule states that an agent must perform an action before its applicability condition becomes false; a permission rule establishes that the agent can bring about ϕ if its condition(s) is true. We show two simple examples of how to model interaction protocols of the FIPA IPL in terms of the rights of the agents to use the performatives. In the FIPA interaction protocol for query-if, agent X queries agent Y whether or not a proposition P is true. The receiver has the right to either agree or refuse to send and *inform* message providing an answer. In the case that agent Y agrees, then it has an obligation to send a notification, which can be an inform speech act stating the truth or falsehood of the proposition P. If agent Y sends a refuse message the protocol ends there. We can complement this by specifying the roles of the participating agents as follows:

```
role(X, customer).
role(Y, seller).

right(X, query-if(X, Y, P), _).

right(Y, agree(Y, X, P));
right(Y, refuse(Y, X, P)) :-
   receive(query-if(X, Y, P).
```

```
right(Y, inform(Y, X, P));
right(Y, inform(Y, X, not P)) :-
   send(agree(Y, X, P), _).
```

Policies can then be defined to constrain the agents' use of the speech acts in virtue of their content. For example, agent Y, acting on behalf of an airline company serving flights to European countries, could have a policy that states that it should agree to every query regarding flight tickets to Europe (i.e. answering about flight times and providing the best offer for a potential buyer) and another one specifying that it has the obligation to refuse every query about flights to non European countries.

```
role(X, customer).
role(Y, seller).

obligation(Y, agree(Y, X, P) :-
    receive(query-if(X, Y, P)), europeanFlight(P)).

obligation(Y, refuse(Y, X, P) :-
    receive(query-if(X, Y, P), nonEuropeanFlight(P)).
```

Similarly, other conversation policies can be defined to state that an agent can deceive, or that it has the right to do so in particular circumstances. It can be specified that an agent X will always answer every message it receives etc. The unified framework for agent communication presented in this paper, consisting of two specification languages and the semantic and pragmatic component of an ACL, captures all the requirements states for agent communication languages. Crucially, it establishes the need for defining a pragmatic theory that help agents to use the speech acts available depending on the contextual parameters of a specific scenario. Therefore, conversation policies would play a crucial part in helping agents to achieve the perlocutionary acts of the speech acts used and, as a consequence, fulfil the requirements regarding the desiderata of an ACL for open MAS. Importantly, the conversation policies proposed are based on the meaning of the speech acts defined, so that the high-level character of the ACL is preserved.

6 Concluding Remarks

Our approach analyses agent communication from a perspective in which the meaning of speech acts results from the combination of the semantic and pragmatic specifications. Every speech act is used in a specific context, and that context usually affects agents' communicative behaviour. We propose a normative pragmatics to stabilize interaction in agent communication. There have been quite a lot of work in the specification of ACL semantics [3,17,4,5] while other authors have proposed several interaction protocols to define the order in which messages are used [3,5,7,18].

Two specification languages $MCTL_I$ and $NCTL_I$, provide the semantics for the ACL semantics and pragmatics. Unlike most of other approaches, our specification languages are grounded in a computational model. This would facilitates the compliance

testing of the ACL [10]. We have shown how our proposal can be used to define pragmatic rules using a declarative language. Unlike other approaches, normative pragmatics does not consist only in establishing the order of messages. By defining normative policies, we facilitate the fulfilment of the perlocutionary effects. In this sense, our unified ACL performs better with respect to the requirements discussed in Table 1 than the other proposals.

The characterization of roles is inspired by the literature on organizational concepts [19,15] and adapted for their use in ACL pragmatics. Other authors [20], have also presented temporal deontic logic with dynamic operators. We believe that the combination of deontic, dynamic and temporal notions results in highly complex logics. $NCTL_I$ is far simpler that those formalisms and therefore is easier to axiomatize. Our proposal is also related to [21], but there are several differences: firstly, they do not provide a formal definition for any of the deontic operators they use; secondly, they claim that policies are independent of the ACL semantics, and that in fact policies should be specified in the general structure of the system; thirdly, the use of obligations produces policies that in some cases could be too restrictive for autonomous agents; fourthly, they use an ontology language based on OWL as the policy specification language, but we believe that logic is a more suitable language to reason about multi-agent systems.

For standardization reasons, our proposal intends to be as close as possible to the FIPA ACL specification. With this purpose in mind, we have provided definitions for the actions absent in the FIPA Communicative Actions Library, commissives and declaratives. We understand that in FIPA CAL some of the definitions are unnecessarily complex. This is partially due to the multimodal language used as the semantic language. Current and future work involves the development of protocols and policies using Prolog and studying different verification techniques based on existing temporal logic algorithms.

References

1. Searle, J.R.: Speech Acts. An Essay in the Philosophy of Language. Cambridge: Cambridge University Press (1969)
2. Finin, T., Fritzson, R., McKay, D., McEntire, R.: KQML as an Agent Communication Language. In Adam, N., Bhargava, B., Yesha, Y., eds.: Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94), Gaithersburg, MD, USA, ACM Press (1994) 456–463
3. FIPA ACL: FIPA Communicative Act Library Specification (2002) http://www.fipa.org/repository/aclspecs.html.
4. Singh, M.P.: A social semantics for agent communication languages. In: Dignum, F., Greaves, M. eds.: Issues in Agent Communication. Lectures Notes in Artificial Intelligence, Vol. 1916. Springer-Verlag, Heidelberg (2000)
5. Fornara, N., Colombetti, M.: A commitment-based approach to agent communication. Applied Artificial Intelligence an International Journal **18** (2004) 853–866
6. Greaves, M., Holmback, H., Bradshaw, J.: What is a Conversation Policy? In Dignum, F., Greaves, M., eds.: Issues in Agent Communication. Lectures Notes in Artificial Intelligence, Vol. 1916. Springer-Verlag, Heidelberg (2000) 118–131
7. Pitt, J., Mamdani, A.: A protocol-based semantics for an agent communication language. In: Proceedings of the 16th International Joint Conference on Artificial Intelligence IJCAI'99, Stockholm, Morgan-Kaufmann Publishers (1999) 486–491

8. Emerson, E.A.: Temporal and modal logic. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science, volume B. North Holland, Amsterdam (1990) 995–1072
9. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about Knowledge. The MIT Press, Cambridge, MA (1995)
10. van der Hoek, W., Wooldridge, M.: Towards a logic of rational agency. Logic Journal of the IGPL **11** (2003) 135–159
11. Kripke, S.: Semantical considerations on modal logic. Acta Philosophical Fennica **XVI** (1963) 83–94
12. Cohen, P., Levesque, H.: Communicative actions for artificial agents. In Bradshaw, J.M., ed.: Software Agents. AAAI Press / The MIT Press, Cambridge (MA) (1997)
13. Rao, A.S., Georgeff, M.P.: Decision procedures for bdi logics. Journal of Logic and Computation **8** (1998) 293–342
14. Castelfranchi, C.: Practical permission: Dependence, power and social commitment. In: Proceedings of 2nd workshop on Practical Reasoning and Rationality, London (1997)
15. van der Torre, L., Hulstijn, J., Dastani, M., Broersen, J.: Specifying multiagent organizations. In: Proceedings of the Seventh Workshop on Deontic Logic in Computer Science (Deon'2004). Lectures Notes in Artificial Intelligence, Vol. 3065. Springer, Heidelberg (2004) 243–257
16. Lomuscio, A., Sergot, M.: Deontic interpreted systems. Studia Logica **75** (2003)
17. Labrou, Y., Finin, T.: A semantic approach for KQML - a general purpose communication language for software agents. In: Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94). (1994) 447–455
18. Endriss, U., Maudet, N., Sadri, F., Toni, F.: Logic-based communication protocols. In: Advances in Agent Communication. Lectures Notes in Artificial Intelligence, Vol. 2922. Springer-Verlag, Heidelberg (2004) 91–107
19. Ferber, J., Gutknecht, O.: A meta-model for the analysis of organizations in multi-agent systems. In: Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS'98). (1998) 128–135
20. Dignum, F., Kuiper, R.: Combining dynamic deontic logic and temporal logic for the specification of deadlines. In Sprague, J.R., ed.: Proceedings of thirtieth HICSS, Hawaii (1997)
21. Kagal, L., Finin, T., Joshi, A.: A policy language for a pervasive computing environment. In: IEEE 4th International Workshop on Policies for Distributed Systems and Networks. (2003)

Broadening the Semantic Coverage of Agent Communicative Acts

Hong Jiang and Michael N. Huhns

Department of Computer Science and Engineering
University of South Carolina, Columbia, SC 29208, USA
{jiangh, huhns}@engr.sc.edu

Abstract. Communicative acts-based ACLs specify domain-independent information about communication and relegate domain-dependent information to an unspecified content language. This is reasonable, but the ACLs cover only a small fraction of the domain-independent information possible. As a key element of modern ACLs, the set of communicative acts needs to be as complete as possible to allow agents to communicate the widest range of information with agreed-upon semantics. This paper describes a new approach to broaden the semantic coverage of agent communicative acts. It provides agents with the ability to express more of the semantics of human languages and yields a more powerful ACL. We first describe the main meaning categories and semantics for an ACL, which we derive from prior work on speech-act classifications. Next, we prove the resultant semantic coverage. Finally, we present some example applications, which demonstrate that the approach can combine the benefits of the FIPA ACL with Ballmer and Brennenstuhl's speech act classification, resulting in a more expressive and efficient ACL.

Keywords: ACL, Semantics, Communicative Acts, FIPA.

1 Introduction

As a critical element of multi-agent systems and a key to the successful application of agents in commerce and industry, modern agent communication languages (ACLs), such as the FIPA ACL, provide a standardized set of performatives denoting types of communicative actions. Such ACLs have been designed as general purpose languages to simplify the design of multi-agent systems. However, recent research shows that these ACLs do not support adequately all relevant types of interactions. Serrano and Ossowski [1] report a need for new *ad hoc* sets of performatives in certain contexts, which the FIPA ACL does not support. Singh [2] points out that agents from different venders or even different research projects cannot communicate with each other. In [3], Kinny shows that the FIPA ACL has a confusing amalgam of different formal and informal specification techniques whose net result is ambiguous, inconsistent and underspecified communication. He proposes a set of requirements and desiderata against which an ACL specification can be judged, and briefly explores some of the shortcomings of the FIPA ACL and its original design basis.

M. Kolp et al. (Eds.): AOIS 2005, LNAI 3529, pp. 32–47, 2006.

Early work on communicative act-based ACLs, such as KQML and FIPA, separated the communication problem into three layers—a message transport layer providing the mechanics of a communication, a domain-independent layer of communication semantics, and a domain-dependent content layer. The ACL speech acts were intended to describe the domain-independent middle layer. The problem is that the 22 communicative acts in the current FIPA ACL cover only a small fraction of the domain-independent concepts that an agent might want to express. For example, one agent can `inform` another of a domain concept using the FIPA ACL, but cannot `promise` another something. If an agent wants to make a promise, its only recourse is to express it in the content language, for which there typically is no standardized support.

Therefore, a larger set of communicative acts would be desirable in an ACL to improve understanding among agents. Recognizing that the ~4800 speech acts in [4] would be desirable but impractical to use individually, we describe a feasible approach to broaden the semantic coverage of ACLs by formalizing speech act categories that subsumes the ~4800, enabling the meanings of all the speech acts to be conveyed. Different from [5], we focus on the standard messages used for communication instead of designing a conversation protocol.

Specifically, Section 2 describes prior work on a comprehensive classification of speech acts by Austin, Searle, and Ballmer and Brennenstuhl. The main meaning categories and their semantics are given in Section 3, where we use FIPA's formal semantic language to represent the semantics of our communicative act categories. This permits our approach to combine the benefits of the FIPA ACL with a broader set of communicative acts. Finally, Section 4 proves the semantic coverage by comparing it with the FIPA ACL, and several example applications are described in Section 5.

2 Research Background

Current ACLs derive their language primitives from the linguistic theory of speech acts, originally developed by Austin [6]. The most important part of his work was to point out that human natural language can be viewed as *actions* and people can perform things by speaking. Austin also classified illocutionary acts as verfictives, exercitives, commissives, behabitives and expositives [6]. The classification has been criticized for overlapping categories, too much heterogeneity in categories, ambiguous definitions of classes, and misfits between the classification of verbs and the definition of categories [4,7,8].

Austin's work was extended by Searle [7,9,10,11], who posited that an illocutionary speech act forms the minimum meaningful unit of language. He classified speech acts into five categories: assertives, directives, commissives, declaratives, and expressives. Searle's speech act theory focusses on the speaker. The success of a speech act depends on the speaker's ability to perform a speech act that should be understandable and successful.

Ballmer and Brennenstuhl [4,8] criticize six aspects of Searle's classification: clarity, definition of declaratives as a speech act type, principles used in the classification, selection of illocutionary verbs, vague definition of the illocutionary

point, and vagueness of the line between illocutionary force and propositional content. They propose an alternative classification, which contains both simple linguistic functions such as expression and appeal, and more complex functions such as interaction and discourse. Models for alternative actions are formed and verbs are classified according to the phases of the model.

Ballmer and Brennenstuhl's classification has motivated us to rethink the speech acts used in ACLs. Since the classification is based on an almost complete domain (˜4800 speech acts) and the authors claim they provide a "theoretically justified" classification "based explicitly and systematically on linguistic data", we believe that to generate a speech act set for ACLs based on their classification will be a powerful way to represent meaning. However, this classification is not perfect: the classification for English is obtained by translating the verbs of a German classification, the names of the categories are not systematically chosen, and there is no formal semantic representation for the categories. However, most of these problems can be fixed by rebuilding the categories. Thus, we endeavour herein to derive a reasonable set of categories for agent communication from their theory and to give a formal semantics using more typical English names.

3 Semantic Description

This section describes the semantic categories for a relatively complete set of speech activity verbs, derived from the classification in [4]. The categories reflect an ontological and a conceptual structuring of linguistic behaviour. The main categories and their relationships are represented in Fig. 1. The top node, Speech Acts, represents the entire set of speech acts in human language and the four major groups—Emotion Model, Enaction Model, Interaction Model and Dialogic Model—represent four basic functions of linguistic behaviour.

The *Emotion Model* is the most speaker-oriented and focusses on representing kinds of emotional states of a human or agent.

The *Enaction Model* is a function directed toward a hearer, by which a speaker tries to control the understanding of the hearer.

The *Interaction Model* is a function involving speaker and hearer in mutual verbal actions. This group includes three sub-categories to represent different degrees of the mutual competition: (1) in the Struggle Model, the speaker tries to get control over the hearer, or the speaker is more competitive in controlling mutual verbal actions; (2) in contrast, the hearer is more competitive in the Valuation Model; and (3) in the Institutional Model, the hearer and speaker are equally competitive.

The *Dialogic* Model covers a kind of reciprocal cooperation where there is a better-behaved and more rigidly organized verbal interaction. Its three sub-categories focus on different types of the content and the organization: (1) the Discourse Model focusses on the organization and types of discourse; (2) the Text Model focusses on the textual assimilation and processing of reality, i.e. the specific knowledge involved; (3) and the Theme Model focuses on the process of thematic structuring and its results, in other words, the structure or organization of some knowledge system.

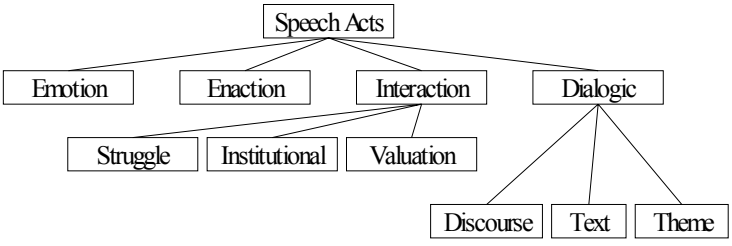

Fig. 1. Ontology of the Main Speech Act Categories

In the above ontology, the four basic models can be divided into unilateral and multilateral models. The Emotion Model and Enaction Model are unilateral, because they focus on a single speech action. The Interaction Model and Dialogic Model are multilateral, because they consider the response from a hearer. The Emotion Model and Interaction Model are more original and less constrained, and the Enaction Model and Dialogic Model are more institutionalized and controlled. Practically, these four basic models may be combined.

We next define several formal semantic model notations and then describe the detailed semantics for the meaning categories.

3.1 Formal Semantic Model Notations

The semantic model used in representing the categories in this paper follows the formal semantic language described for the FIPA ACL [12]. Components of the formalism are

- $p, p_1, ...$ are closed formulas denoting propositions;
- ϕ, ψ are formula schemes, which stand for any closed proposition;
- i, j are schematic variables denoting agents.

The mental model of an agent is based on four primitive attitudes: belief (what the agent knows or can know); desire (what the agent desires); intention (an agent's persistent goal that could lead to some actions); and uncertainty. They are respectively formalized by operators B, D, I, and U:

- $B_i p$ agent i (implicitly) believes (that) p;
- $D_i p$ agent i desires that p currently holds;
- $I_i p$ agent i intends a persistent goal p;
- $U_i p$ agent i is uncertain about p, but thinks that p is more likely than $\neg p$;

We use the abbreviations:

- $Bif_i\phi \equiv B_i\phi \vee B_i\neg\phi$, which means that agent i believes either ϕ or $\neg\phi$.
- $Uif_i\phi \equiv U_i\phi \vee U_i\neg\phi$, which means that either agent i is uncertain about ϕ (ϕ is more likely) or $\neg\phi$ ($\neg\phi$ is more likely).

To support reasoning about action, we also introduce operators *Feasible*, *Done* and *Agent*:

- $Feasible(a, p)$ means that an action a can take place and, if it does, then p will be true.
- $Done(a, p)$ means that when p is true, then action a takes place.
- $Agent(i, a)$ means agent i is the agent who performs action a.

Generally, the components of a speech act model involved in a planning process should contain both the conditions that have to be satisfied for the act to be planned and the reasons for which the act is selected. The former is termed FP (feasibility preconditions) and the latter RE (rational effect). We use the same model here, represented as

$$\begin{aligned} &< i, act\ (j, C) > \\ &FP : \phi_1 \\ &RE : \phi_2 \end{aligned} \tag{1}$$

where i is the sender or speaker, j the recipient or hearer, act is the name of the speech act, C is the semantic content, and ϕ_1 and ϕ_2 are propositions.

3.2 Emotion Model

The Emotion Model focusses on representing the emotional states of a human or agent. We assume there is a finite set of emotions, E, represented as

$$E = \{e_+, e_0, e_-\} \tag{2}$$

where e_+ is an emotion in the set of positive emotions, which is characterized by or displaying a kind of certainty, acceptance or affirmation (about the content involved), such as $\{happy, love, ...\}$; e_0 is in the set of neutral emotions, which does not show any tendency, such as $\{hesitate, ...\}$; e_- is in the set of negative emotions, which intends or expresses a kind of negation, refusal or denial, such as $\{angry, sad, afraid, ...\}$.

The Emotion Model is represented as follows:

$$\begin{aligned} &< i,\ em\ \ (j, \phi) > \\ &FP : \neg B_i\ (B_j Agent(i, em(\phi))) \wedge D_i(B_j Agent(i, em(\phi))) \\ &RE :\ B_j\ \ Agent(i, em(\phi)) \end{aligned} \tag{3}$$

where $em \in E$ and the semantic content ϕ can be empty. Here, desire D is used instead of the stronger notion I, since emotions are easy to show for humans. This model represents that agent i sends a message to j that i has emotion em about ϕ or i is in the state of em when ϕ is empty. The FP shows that, when agent i does not believe that agent j knows that i is currently in emotion em about ϕ, and i desires that j knows it, then this message can be sent. The RE shows that the desired result is that agent j believes that i is in emotion em about ϕ.

To simplify usage of this model, we can directly use e_+, e_0, or e_- as communicative acts. In this case, we focus on the effect of the emotion speech act on the

content ϕ. That is, for a positive effect, i hopes j knows that i has an intention on ϕ; for a negative one, i hopes j knows that i has a negative intention on ϕ; for a neutral one, i shows its attitude is uncertain about ϕ. Just like human interactions, we do not have to know the precise value of an attitude. Instead, we just need to know that something is viewed favourably, unfavourably or neutrally.

However, detailed emotions are also desirable in some cases. To make this usable, we generate a set of foundational meaning units from 155 emotion speech acts listed in [4]. Table 1 gives the foundational meaning units of emotions that combine the idea from [13,14], and they are organized with consideration of positive, neutral and negative values.

Table 1. Foundational Meaning Units of Emotional Speech Acts

+	0	-
happy	N/A	sad
love	N/A	hate
excited	nervous	angry
desire	hesitate	fear
N/A	shocked	N/A

In Table 1, each row represents a kind of meaning unit. In the first row, *sad* has the opposite meaning of *happy*. *Hate* has the opposite meaning of *love* in the second row. *Excited* represents a positive attitude to something with strong feeling, *nervous* represents a strong uncertain feeling about something and *angry* represents a strong negative feeling about something. In the fourth row, *desire* shows a feeling to get something, *hesitate* shows no intentions or some uncertainty and $fear$ shows a feeling to avoid something. In the last row, *shocked* shows a neutral feeling about surprise.

3.3 Enaction Model

In the Enaction Model, the speaker more or less coercively attempts to get the hearer to do something by expressing an idea, wish, intention, proposal, goal etc. There are many speech acts in this group. To organize them and simplify the usage, we define the set of enactions as:

$$EN = \{en_+, en_-\} \tag{4}$$

Unlike the Emotion Model, which describes emotions, the Enaction Model tries to make a hearer do something. Thus, there are no neutral enactions: if agent i does not want j to do anything, i does not have to send any message to j. en_+ is an action in the set of positive enactions, such as $\{intend, desire, askfor$, *encourage*, ...$\}$; en_- is an action in the set of negative enactions, such as $\{$ *warning*, *cancel*, ...$\}$.

The Enaction Model can be defined as:

$$\begin{aligned} & < i,\ en_{\pm}(j,\phi) > \\ FP:\ & \neg B_i\phi \wedge D_i\phi \wedge B_i(B_j\phi \wedge \neg D_j\phi) \qquad for\ en_+ \\ & \neg B_i\neg\phi \wedge D_i\neg\phi \wedge B_i(B_j\neg\phi \wedge \neg D_j\neg\phi) \quad for\ en_- \\ RE:\ & Done(en_{\pm}(\phi)) \end{aligned} \tag{5}$$

where $en_{\pm} \in EN$. This model represents agent i sending a message to j to ask j to do $en_{\pm}$ on ϕ. The FP shows that this message could be sent for en_+ when i does not believe that i can do ϕ and it desires ϕ, while i believes that j can do it, but j does not want to do it. FP is the same for en_-, except ϕ is replaced by $\neg\phi$. The expected result is that $en_{\pm}$ on ϕ is done. Practically, j could just add the action to its action queue for a positive enaction (in this case, $Done(en_+(\phi)) = Done(\phi)$) or delete it from its queue for a negative enaction.

3.4 Interaction Model

The Interaction Model is a function involving a speaker and a hearer mutually interacting. We assume an interaction set IN and the communicative act set $Acts$ so that $IN \subseteq Acts$, and for some $in \in IN$ and $act \in Acts$, $\exists rule : in \rightarrow act$, such that:

$$\begin{aligned} & < i,\ in(j,(a,goal)) > \\ FP:\ & I_i goal \wedge \neg B_i Done(a) \wedge D_i Done(a) \wedge B_i(Agent(j,a) \wedge \neg D_j Done(a)) \\ RE:\ & Done(a) \wedge (< j, act(i,(a', goal - a) > \vee < j, succeed(i, goal) >) \\ & \vee < j, fail(i, goal) > \end{aligned} \tag{6}$$

where a, a' are actions, and $goal$ can be a plan or a sequence of actions. This model represents agent i sending a message to j to ask j to do action a for some $goal$. The FP shows that i intends to achieve the $goal$, so i desires to do a but cannot do it itself, and i believes that j can do it. However, j does not desire to do it. The expected result is j does a first, and then generates another message back to i. This reply message follows the rule $in \rightarrow act$. Generally, the message has the form $< j, act(i,(a', goal - a) >$, which means that after j has done a, it generates another action a' and reduces the $goal$. In some cases, for example after j has done a and the $goal$ is already achieved, then j sends back the message $< j, succeed(i, goal) >$, which means that the $goal$ is achieved. Another extreme case is that j finds out that the $goal$ is impossible to be achieved, then it sends back message $< j, fail(i, goal) >$, which means the $goal$ is unachievable.

There are three subcategories of the interaction model representing different degrees of the mutual competition: Struggle Model, Institutional Model and Valuation Model. In the Struggle Model, the speaker tries to get control over the hearer or the speaker is more competitive in controlling mutual verbal actions. In this case, the rule $in \rightarrow act$ is decided by the speaker or sender i.

In the Institutional Model, the hearer and speaker are equally competitive. For example, the establishment of a behaviour in an institution equally affects the

upholders of and the participants in the institution, especially when entering an institution and thereby adopting its norms, following its norms and rules, violating them and being pursued by the upholders of the institution. Thus, the agents i and j should have some common rule system defined in advance.

In the Valuation Model, the hearer is more competitive, so it decides which communication act to use in its reply. That is, the rule $in \rightarrow act$ is decided by agent j after its evaluation of the previous message. Details of the Valuation Model cover both positive and negative valuations of actions, persons, things and states of affairs.

3.5 Dialogic Model

The Dialogic Model covers a kind of reciprocal cooperation, and is a more regular and constrained verbal interaction. For this model, we at first assume a dialogic speech act set DS and the communicative act set $Acts$ so that $DS \subseteq Acts$, and for some $d \in DS$ and $act \in Acts$, $\exists rule : d \rightarrow act$, such that

$$\begin{aligned} &< i,\ d(j, \phi) > \\ FP:\ &B_i\phi \wedge D_i B_j \phi \\ RE:\ &B_j\phi \wedge < j, act(i, \phi') > \end{aligned} \tag{7}$$

For agent i to send a message to j about ϕ in this model, agent i believes ϕ, and i desires j to believe it. The expected result is that j believes ϕ and j replies to i with another message about a new ϕ, which is the reasoning result of agent j, and the communicative act used in the message follows the rule $d \rightarrow act$.

Corresponding to the three subcategories that focus on different types of content and organization, we can define three types for ϕ:

- The Discourse Model focusses on the organization and types of discourse, so ϕ points to some kind of type or organization that is predefined. For example, according to the status of a discourse, it could be { *beginning discourse, being in discourse, discourse inconvenience, reconciliation of discourse, ending discourse* }; according to the attitude for some content, it could be { *accept, refuse, cancel* }; according to the number of agents involved in the discourse, it could be { *discourse with several speakers, discourse with one speaker, ...* }; or a kind of irony, joke etc.
- The Text Model focusses on the textual assimilation and processing of the specific knowledge involved, i.e. ϕ describes some knowledge about perceiving reality, producing texts, systematically searching for data etc.
- The Theme Model focusses on the process of thematic structuring and its results, in other words, ϕ points to the structure or organization of some knowledge system.

4 Proof of Semantic Coverage

The FIPA ACL has four primitive communicative acts, and its other communicative acts are composed of the primitive acts or are composed from primitive messages by substitution or sequencing [12]. The four primitive acts are:

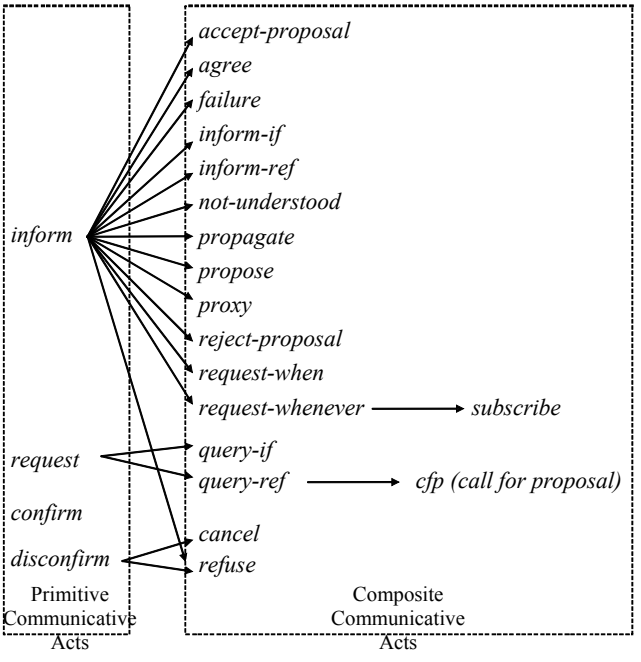

Fig. 2. Relationship of FIPA Primitive and Composite Communicative Acts

– The Assertive Inform:

$$\begin{aligned} &< i,\ inform(j,\phi) > \\ FP:\ &B_i\phi \wedge \neg B_i(Bif_j\phi \vee Uif_j\phi) \\ RE:\ &B_j\phi \end{aligned}$$

– The Directive Request:

$$\begin{aligned} &< i,\ request(j,a) > \\ FP:\ &FP(a)[i\backslash j] \wedge B_i Agent(j,a) \wedge B_i \neg PG_j Done(a) \\ RE:\ &Done(a) \end{aligned}$$

where $FP(a)$ denotes the feasibility preconditions of a; $FP(a)[i\backslash j]$ denotes the part of the FPs of a that are mental attitudes of i; and PG_iP means that i has P as a persistent goal.

– Confirming an Uncertain Proposition (Confirm):

$$\begin{aligned} &< i,\ confirm(j,\phi) > \\ FP:\ &B_i\phi \wedge B_i U_j\phi \\ RE:\ &B_j\phi \end{aligned}$$

– Contradiction Knowledge (Disconfirm):

$$\begin{aligned} &< i,\ \mathit{disconfirm}(j,\phi) > \\ FP:&\ B_i\neg\phi \wedge B_i(U_j\phi \vee B_j\phi) \\ RE:&\ B_j\neg\phi \end{aligned}$$

Furthermore, among the 22 communicative acts of FIPA ACL, the composite ones corresponding to the above four primitive acts are as shown in Fig. 2:

– Inform: accept-proposal, agree, failure, inform-if, inform-ref, not-understood, propagate, propose, proxy, reject-proposal, request-when, request-whenever, subscribe
– Request: cfp(call for proposal), query-if, query-ref
– Confirm: N/A
– Disconfirm: cancel, refuse

It can be seen that the composite communicative acts relate to the primitive ones unevenly. Most of the communicative acts are derived from *inform*, and even the primitive acts, *confirm* and *disconfirm*, are special cases of *inform*, which can be proved as follows.

Lemma 1. *In the primitive communicative acts of FIPA ACL, confirm (*$< i, \mathit{confirm}(j,\phi) >$*) is a special case of inform (*$< i, \mathit{inform}(j,\phi) >$*).*

Proof. Comparing the definitions of *confirm* and *inform*, we see they have the same message body format and rational effect—*RE*. The only difference is the feasibility preconditions—*FP*. We can then try to prove that *FP* of *confirm* is a sufficient but not necessary condition of *inform*. That is, when the *FP* of *confirm* is satisfied, the *FP* of *inform* is also satisfied, or the satisfaction of *FP* of *confirm* can trigger an *inform* message; alternatively, the *FP* of *confirm* is not necessary for sending an *inform*.

FP of *inform* is:

$$\begin{aligned} & B_i\phi \wedge \neg B_i(Bif_j\phi \vee Uif_j\phi) \\ \equiv\ & B_i\phi \wedge \neg B_i((B_j\phi \vee B_j\neg\phi) \vee (U_j\phi \vee U_j\neg\phi)) && (8) \\ \equiv\ & B_i\phi \wedge (\neg B_iB_j\phi \vee \neg B_iB_j\neg\phi \vee \neg B_iU_j\phi \vee \neg B_iU_j\neg\phi) \\ \equiv\ & B_i\phi \wedge (\neg B_iB_j\phi \vee B_iB_j\phi \vee \neg B_iU_j\phi \vee B_iU_j\phi) && (9) \\ \equiv\ & (B_i\phi \wedge \neg B_iB_j\phi) \vee (B_i\phi \wedge B_iB_j\phi) \vee (B_i\phi \wedge \neg B_iU_j\phi) \vee (B_i\phi \wedge B_iU_j\phi) && (10) \end{aligned}$$

where equation (8) is derived from the definitions of Bif_ip and Uif_ip. We get equation (9) since agent i not believing j believes ϕ usually means the same as agent i believing j does not believe ϕ.

From equation (10), the last part $B_i\phi \wedge B_iU_j\phi$ is exactly the *FP* of *confirm*. When *FP* of *confirm* is satisfied, that is, when $B_i\phi \wedge B_iU_j\phi$ is true, then equation (10) will also be true. However, *FP* of *confirm* is not a necessary condition, since only if one of $B_i\phi \wedge \neg B_iB_j\phi$, $B_i\phi \wedge B_iB_j\phi$ and $B_i\phi \wedge \neg B_iU_j\phi$ is satisfied, equation (10) will also be satisfied.

Thus, *confirm* is a special case of *inform*.

Lemma 2. *In the primitive communicative acts of the FIPA ACL, disconfirm* $(< i, disconfirm(j, \phi) >)$ *is a special case of inform* $(< i, inform(j, \neg\phi) >)$.

Proof. Comparing the definitions of *disconfirm* and *inform*, we have

$$\begin{aligned} & < i,\ inform(j, \neg\phi) > \\ FP:\ & B_i\neg\phi \wedge \neg B_i(Bif_j\neg\phi \vee Uif_j\neg\phi) \\ RE:\ & B_j\neg\phi \end{aligned} \tag{11}$$

Thus, we get the same rational effect format—RE. Let's compare the feasibility preconditions—FP, and similarly equation (11) can be changed to:

$$\begin{aligned} & B_i\neg\phi \wedge \neg B_i(Bif_j\neg\phi \vee Uif_j\neg\phi) \\ \equiv\ & B_i\neg\phi \wedge \neg B_i((B_j\neg\phi \vee B_j\phi) \vee (U_j\neg\phi \vee U_j\phi)) \\ \equiv\ & B_i\neg\phi \wedge \neg B_i(B_j\neg\phi \vee B_j\phi \vee U_j\neg\phi \vee U_j\phi) \\ \equiv\ & B_i\neg\phi \wedge \neg B_i((U_j\phi \vee B_j\phi) \vee (B_j\neg\phi \vee U_j\neg\phi)) \\ \equiv\ & B_i\neg\phi \wedge (\neg B_i(U_j\phi \vee B_j\phi) \vee \neg B_i(B_j\neg\phi \vee U_j\neg\phi)) \\ \equiv\ & (B_i\neg\phi \wedge \neg B_i(U_j\phi \vee B_j\phi)) \vee (B_i\neg\phi \wedge \neg B_i(B_j\neg\phi \vee U_j\neg\phi)) \end{aligned} \tag{12}$$

From equation (12), the first part $B_i\neg\phi \wedge \neg B_i(U_j\phi \vee B_j\phi)$ is exactly the FP of *disconfirm*. When FP of *disconfirm* is satisfied, that is, when $B_i\neg\phi \wedge \neg B_i(U_j\phi \vee B_j\phi)$ is true, then equation (12) will also be true, which will trigger message $< i, inform(j, \neg\phi) >$. However, FP of *disconfirm* is not a necessary condition, since if $B_i\neg\phi \wedge \neg B_i(B_j\neg\phi \vee U_j\neg\phi)$ is satisfied, equation (12) will also be satisfied.

Thus, we proved that FP of *disconfirm* is a sufficient but not necessary condition to trigger message $< i, inform(j, \neg\phi) >$. In other words, *disconfirm* $(< i, disconfirm(j, \phi) >)$ is a special case of *inform* $(< i, inform(j, \neg\phi) >)$.

So far, we can conclude that there are actually two foundational communicative acts *inform* and *request*. If we can prove that our approach covers the semantic meaning of these two communicative acts, then our approach covers all the semantic meanings of the FIPA communicative acts, since the others can be derived from these two by adding constraints.

However, we think *inform* is too general. Considering $\neg B_i(Bif_j\phi \vee Uif_j\phi)$ in FP of *inform*, it actually lists all the possibility of j's knowledge about ϕ, such that: i does not believe j believes ϕ, or i does not believe j believes not ϕ, or i does not believe j is uncertain about ϕ, or i does not believe j is uncertain about not ϕ. Since at least one of them will be true, $\neg B_i(Bif_j\phi \vee Uif_j\phi)$ will always be true. So FP of *inform* can be simplified to $B_i\phi$, which is reasonable because only if agent i has the belief ϕ can it inform j about ϕ. While, we still think it should not ignore the desire to have j to believe ϕ, no matter what i believes or does not believe j's knowledge about ϕ, if i does not have any desire to have j believe ϕ, why does i want to send the message to j?

Based on the above analysis, now we prove that our approach covers the semantic meaning of the two foundational communicative acts *inform* and *request*.

Lemma 3. *The Dialogic model covers the semantic meaning of FIPA's inform.*

Proof. According to our above analysis of *inform*, $\neg B_i(Bif_j\phi \vee Uif_j\phi)$ did not supply any of i's opinion on j's knowledge about ϕ, and i's desire for j to know about ϕ was also be ignored. By adding these considerations, we can then represent *inform* with more precise semantic meaning as:

$$\begin{aligned} &< i,\ inform(j,\phi) > \\ FP:&\ B_i\phi \wedge D_iB_j\phi \\ RE:&\ B_j\phi \end{aligned}$$

Then it has a format similar to the semantic representation of the dialogic model, and the difference is in RE. For the dialogic model, we assume a dialogic communicative act set DS and the communicative act set $Acts$ with $DS \subseteq Acts$, and for some $d \in DS$ and $act \in Acts$, $\exists rule : d \rightarrow act$, If $d \in DS$ is a terminal symbol, that is, there is no rule from d to something else, then in this case, $< j, act(i,\phi') >$ in RE can be ignored, so that we can get the same semantic meaning of *inform*. Thus, we can use the *dialogic model* to represent the semantic meaning of *inform*.

Lemma 4. *The interaction model covers the semantic meaning of FIPA's request.*

Proof. Let's first consider the definition of *request*, which is used to request a receiver to perform some action. Usually it presumes feedback from the receiver. FP of *request* involves three parts:

- $FP(a)[i\backslash j]$: denotes the part of the FPs of action a that are mental attitudes of agent i. We do not know exactly what the mental attitudes will be, although they should satisfy the following conditions for sending out a request: agent i should intend to have action a done—$I_iDone(a)$; and i cannot do a by itself.
- $B_iAgent(j,a)$: i believes that j is the only agent that can perform a.
- $B_i\neg PG_jDone(a)$: this part (in page 36 of [12]) is also presented as $\neg B_iI_j$ $Done$ (a) (in page 25 of [12]), which roughly points out a required condition: i does not believe j intents to do a.

The *goal* in the interaction model denotes a plan or a sequence of actions. To get comparable format of the interaction model, we can let the goal involve only one action, that is, let $goal = Done(a)$. Then the interaction model can be simplified to:

$$\begin{aligned} &< i,\ in(j,a) > \\ FP:&\ I_iDone(a) \wedge \neg B_iDone(a) \wedge D_iDone(a) \wedge \\ &\quad B_i(Agent(j,a) \wedge \neg D_jDone(a)) \\ RE:&\ Done(a) \wedge < j, succeed(i, Done(a)) > \end{aligned} \tag{13}$$

In the BDI model, intention is generated from desire. If we separate desires into intentional desires (I) and non-intentional desires (NI), then we can represent $D_i p$ to be $I_i p \vee NI_i\, p$, such that equation (13) becomes

$$\begin{aligned}
& I_i Done(a) \wedge \neg B_i Done(a) \wedge D_i Done(a) \wedge B_i(Agent(j,a) \wedge \neg D_j Done(a)) \\
\equiv\ & I_i Done(a) \wedge \neg B_i Done(a) \wedge (I_i Done(a) \vee NI_i\, Done(a)) \\
& \wedge B_i(Agent(j,a) \wedge \neg D_j Done(a)) \\
\equiv\ & I_i Done(a) \wedge \neg B_i Done(a) \wedge B_i(Agent(j,a) \wedge \neg D_j Done(a)) \\
\equiv\ & I_i Done(a) \wedge \neg B_i Done(a) \wedge B_i Agent(j,a) \wedge B_i \neg D_j Done(a) \\
\equiv\ & I_i Done(a) \wedge \neg B_i Done(a) \wedge B_i Agent(j,a) \wedge \neg B_i(I_i Done(a) \vee NI_i\, Done(a)) \\
\equiv\ & (I_i Done(a) \wedge \neg B_i Done(a) \wedge B_i Agent(j,a) \wedge \neg B_i I_i Done(a)) \vee \\
& (I_i Done(a) \wedge \neg B_i Done(a) \wedge B_i Agent(j,a) \wedge \neg B_i\, NI_i\, Done(a)) \qquad (14)
\end{aligned}$$

The first part of equation (14) has a format similar to the *FP* of *request*:

- $I_i Done(a) \wedge \neg B_i Done(a)$ corresponds to the first part of *FP* for *request*, which presents the detailed required conditions—agent i should intend to have action a done and i cannot do a by itself.
- $B_i Agent(j,a)$ is the same as the second part of *FP* for *request*.
- $\neg B_i I_i Done(a)$ is the same as the third part of *FP* for *request*. We did not use symbol *PG* in our approach, since *PG* is very similar to I, and here this part follows the format on page 25 of [12].

So far, we see that when *FP* of *request* is true, the equation (14) will also be true, and message of *interaction model* will be triggered. However, the *FP* of *request* is not necessary for equation (14) to be satisfied.

Let's continue to consider the *RE* of *request*, which is the same as the first part of *RE* of *interaction model*. However, the second part is also reasonable for *request*, since in most cases *request* implies feedback from the receiver.

Thus, *request* is a special case of *interaction model*, and the *interaction model* covers the semantic meaning of *request* in the FIPA ACL.

In summary, our approach covers the semantic meaning of the two foundational communicative acts, so it also covers all the semantic meanings of the communicative acts in the FIPA ACL. Moreover, our approach also covers additional semantic meanings. For example, our emotion model supplies a way to communicate emotions, which the FIPA ACL does not. We believe it is important to cover emotions in agent communicative acts, since other researchers [15,16,17] have already discovered that emotions influence human decision-making; unfortunately, this influence has traditionally been ignored.

5 Example Applications

This section provides several examples showing how these defined semantic categories can be used.

Example 1: Bob tells Sue that he loves her. Using the emotion model, the sender is Bob, the receiver is Sue and ϕ = Sue to yield the message on the left below. The expected result will be that Sue has a belief that Bob is in love with her. Since the FIPA ACL does not have a communicative act with a similar meaning, the content must include the expression of emotion, as shown in the message on the right.

```
(love                          (inform
:sender Bob                    :sender Bob
:receiver Sue                  :receiver Sue
:content (Sue))                :content (Bob loves Sue))
```

The left message separates domain independent from domain dependent information better and is less ambiguous.

Example 2: Jack commands Bill to turn off the TV. Using the enaction model, the message to be sent is

```
(command
:sender Jack
:receiver Bill
:content (turn off the TV))
```

The expected result will be that Bill turns off the TV. The communicative act *command* implies a master-slave relationship between the sender and receiver. The FIPA ACL does not have a similar communicative act, so all the information must be put in the content, as in ***Example 1***, although it is more ambiguous.

Example 3: Bob and Jack work together to open a case with ID 011. Bob gets the key but it is broken. Jack is an expert in fixing keys, so Bob asks Jack to fix the key.

According to the interaction model, the message sent to Jack will be

```
(interact
:sender Bob
:receiver Jack
:content (fix key) (open case 011))
```

The goal "open case 011" implies a sequence of actions, which are assumed known to both sender and receiver in advance. Thus Jack tries to fix the key. If Jack fixes the key successfully, he will send a reply to Bob that Bob can pick up the key to open the case now, as shown in the message below on the left. If Jack cannot fix the key, he will then tell Bob that the goal failed, as shown on the right.

```
(interact                      (fail
:sender Jack                   :sender Jack
:receiver Bob                  :receiver Bob
:content (pick-up key)         :content (open case 011)
    (open case 011 - fix key))
```

This model is especially useful for multiple agents working together on a project.

Example 4: Bill wants to tell Bob about the structure of the subway system in Boston, which includes the red line, orange line, green line and blue line. According to the dialogic model, the message sent to Bob would be

(structure
:sender Bill
:receiver Bob
:content (Subway in Boston: red line, orange line, green line, blue line)

The expected result will be that Bob records the structure information as one of his beliefs. We can also use FIPA's $inform$ to represent the above message, but the relationship of the subway system and those lines would have to be part of the content.

6 Conclusion and Future Work

Comparing our approach to the FIPA ACL reveals that:

Better coverage: our approach covers more of human semantics.

Precise semantics: we adopt the same formalism as used by FIPA for our four basic categories and subcategories.

Easy usage: An ACL must be easy to use, and the FIPA ACL has many successful uses. Instead of replacing it, we substitute our communicative acts and keep its message structure. We organize the communicative acts as an ontology with different abstract levels, so that a user or agent can more easily navigate through them to choose the desired ones.

Better understood: Easy usage requires that the ACL be well understood. However, the original categories given by Ballmer and Brennenstuhl's classification are poor, because the classification is obtained by translating German verbs and the names of the categories are not chosen systematically. We modified their classification by using typical English names, which should be more understandable.

Efficiency: Efficiency is desirable for an ACL. As can be seen in the above examples, our approach separates domain-independent from domain-dependent information better, which can shorten the message sent while improving the semantics.

In summary, our approach combines the benefits of the FIPA ACL and Ballmer and Brennenstuhl's speech act classification. It is more expressive in representing a broader range of domain-independent communication semantics, while remaining consistent with current approaches to ACLs. However, a better communicative act set with reasonable size still needs work. Instead of just con-

sidering the categories, some frequently used speech acts also need to be found for the communicative act set.

References

1. Serrano, J.M., Ossowski, S.: An organizational metamodel for the design of catalogues of communicative actions. In: Proceedings of the 5th Pacific Rim International Workshop on Multi Agents, London, UK, Springer-Verlag (2002) 92–108
2. Singh, M.P.: Agent communication languages: Rethinking the principles. Computer **31**(12) (1998) 40–47
3. Kinny, D.: Reliable agent communication - a pragmatic perspective. In: PRIMA '99: Proceedings of the Second Pacific Rim International Workshop on Multi-Agents, London, UK, Springer-Verlag (1999) 16–31
4. Ballmer, T., Brennenstuhl, W.: Speech Act Classification – A Study in the lexical Analysis of English Speech Activity Verbs. Springer-Verlag, Berlin, Heidelberg, New York (1981)
5. Chang, M.K., Woo, C.C.: A speech-act-based negotiation protocol: design, implementation, and test use. ACM Trans. Inf. Syst. **12**(4) (1994) 360–382
6. Austin, J.L.: How to do Things with Words. Oxford University Press, Oxford, UK (1962)
7. Searle, J.R., Vanderveken, D.: Foundations of Illocutionary Logic. Cambridge U. Press, London (1985)
8. Auramaki, E., Lyytinen, K.: On the success of speech acts and negotiating commitments. In: LAP'96: Proceedings of the First International Workshop on Communication Modelling, Oisterwijk, The Netherlands (1996)
9. Searle, J.R.: Speech Acts: An Essay in the Philosophy of Language. Cambridge U. Press, Cambridge, England (1970)
10. Searle, J.R.: A taxonomy of illocutionary acts. Language, Mind, and Knowledge **7** (1975)
11. Searle, J.R.: Expression and Meaning: Studies in the Theory of Speech Acts. Cambridge U. Press, Cambridge (1979)
12. FIPA: FIPA Communicative Act Library Specification. Foundation for Intelligent Physical Agent, Geneva, Switzerland (2002)
13. Li yun (ceremenial usages: Their origin, development, and intention). The Book of Rites (200 B.C.)
14. Ekman, P., Davidson, R.J., eds.: The Nature of Emotion: Fundamental Questions. Oxford University Press, New York (1994)
15. Damasio, A.R.: Descartes' Error : Emotion, Reason, and the Human Brain. Avon Books, New York (1994)
16. Wegner, D.M., Wheatley, T.: Apparent mental causation: Sources of the experience of will. American Psychologist **54**(7) (1999) 480–492
17. Bechara, A.: The role of emotion in decision-making: Evidence from neurological patients with orbitofrontal damage. Brain and Cognition **55**(1) (2004) 30–40

Requirements Analysis of an Agent's Reasoning Capability

Tibor Bosse[1], Catholijn M. Jonker[2], and Jan Treur[1]

[1] Vrije Universiteit Amsterdam, Department of Artificial Intelligence,
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
{tbosse, treur}@cs.vu.nl
http://www.cs.vu.nl/~{tbosse, treur}
[2] Radboud Universiteit Nijmegen, Nijmegen Institute for Cognition and Information,
Montessorilaan 3, 6525 HR Nijmegen, The Netherlands
C.Jonker@nici.ru.nl
http://www.nici.ru.nl/~catholj

Abstract. The aim of requirements analysis for an agent that is to be designed is to identify what characteristic capabilities the agent should have. One of the characteristics usually expected for intelligent agents is the capability of reasoning. This paper shows how a requirements analysis of an agent's reasoning capability can be made. Reasoning processes may involve dynamically introduced or retracted assumptions: 'reasoning by assumption'. It is shown for this type of reasoning how relevant dynamic properties at different levels of aggregation can be identified as requirements that characterize the reasoning capability. A software agent has been built that performs this type of reasoning. The dynamic properties have been expressed using the temporal trace language TTL and can and have been checked automatically for sample traces.

1 Introduction

Requirements analysis addresses the identification and specification of the functionality expected for the system to be developed, abstracting from the manner in which this functionality is realized in a design and implementation of this system; e.g. [1-3]. Recently, requirements analysis for concurrent systems and agent systems has been addressed in particular, for example, in [4, 5]. An agent-oriented view on requirements analysis can benefit from the more specific assumptions on structures and capabilities expected for agents, compared to software components in general. To obtain these benefits, a dedicated agent-oriented requirements analysis process can be performed that takes into account specific agent-related structures and capabilities. For example, for a number of often-occurring agent capabilities, a requirements analysis can be made and documented that is reusable in future agent-oriented software engineering processes. In the process of building agent systems, software engineering principles and techniques, such as scenario and requirements specification, verification and validation, can be supported by the reusable results of such a requirements analysis.

M. Kolp et al. (Eds.): AOIS 2005, LNAI 3529, pp. 48 – 63, 2006.

In this paper the results are presented of a requirements analysis of an agent's reasoning capability. Since reasoning can take different forms, intelligent agents may sometimes require nontrivial reasoning capabilities. The more simple forms of reasoning amount to determining the deductive closure of a logical theory (a knowledge base), given a set of input facts. Requirements for such reasoning processes can be specified in the form of a functional relation between input and output states, abstracting from the time it takes to perform the reasoning e.g. [6]. Properties of such a functional relation can be related to properties of a knowledge base used to realize the functionality, which provides possibilities for verification and validation of this knowledge e.g. [7]. However, more sophisticated reasoning capabilities can better be considered as involving a process over time; especially for nontrivial reasoning patterns when the temporal aspects play an important role in their semantics cf. [8, 9]. Therefore, within an agent-oriented software engineering approach to an agent's reasoning capability, requirements specification has to address the dynamic properties of a reasoning process.

This paper shows how such a requirements analysis of the dynamics of an agent's reasoning capability can be made. The approach makes use of a semantic formalization of reasoning processes by *traces* consisting of sequences of reasoning states over time, following the semantic formalization introduced in [8]. Reasoning processes as performed by humans may involve dynamically introduced or retracted assumptions: a pattern used as a case study in this paper, called 'reasoning by assumption'. For requirements acquisition, it is to be shown for this type of reasoning which relevant dynamic properties can be identified that characterize the reasoning pattern.

For the requirements analysis of an agent's capability to perform this type of reasoning, a methodology has been used that comprises the following steps:

- First, a number of scenarios of practical human reasoning processes considered as 'reasoning by assumption' have been analysed and specified to identify requirements that are characteristic for this reasoning pattern. Required dynamic properties at different levels of aggregation (or grain size) have been identified. These characterizing properties have been formalized using the temporal trace language TTL, thus permitting automated support of analysis.
- The specified dynamic properties at the lowest aggregation level are in an executable format; they specify reasoning steps. Using a variant of Executable Temporal Logic [10] and a dedicated software environment for simulation that has been developed [11], these executable properties were used to generate abstract simulation traces. Such traces can be used to provide system designers with a concrete idea of the intended flow of events over time, without having to actually implement the system.
- Next, logical relationships have been determined between dynamic properties at different aggregation levels, in such a way that the dynamic properties at one aggregation level together imply those at a higher aggregation level. Such logical relationships constitute a formal theory of the interdependencies of the different requirements.

- Finally, verification of the requirements has been performed. Supported by software tools, the dynamic properties at different levels have been checked against three different types of traces: (1) human traces, (2) simulation traces, and (3) prototype traces. As for (1), a number of reasoning puzzles were used to acquire scenarios of further practical human reasoning processes that intuitively fit the pattern of reasoning by assumption [12]. The properties were then automatically checked against the formalized scenarios of these human traces. Concerning (2), the (higher-level) dynamic properties were checked against the traces that resulted from the simulation mentioned above, and confirmed, which validates the identified logical relationships between the dynamic properties at different aggregation levels. Finally, as for (3), a design of an existing software agent performing reasoning by assumption [13] was analysed. This agent was designed using the component-based design method DESIRE [14]. Using the DESIRE execution environment, for this agent a number of reasoning traces were generated. For these traces, all identified dynamic properties (also the executable ones) were also checked and found to be confirmed.

In Section 2 the dynamic perspective on reasoning is discussed in further detail, focussed on the pattern 'reasoning by assumption'. Section 3 addresses some details of the language used. Section 4 presents a number of requirements in the form of dynamic properties identified for patterns of reasoning by assumption. Section 5 discusses logical relationships between dynamic properties at different aggregation levels. In Section 6, it is discussed in which respects verification has been performed. In Section 7, the contribution of the research presented in the paper is briefly discussed.

2 The Dynamics of Reasoning

Analysis of reasoning processes has been addressed from different areas and angles, for example, Cognitive Science, Philosophy and Logic, and AI. For reasoning processes in natural contexts, which are usually not restricted to simple deduction, dynamic aspects play an important role and have to be taken into account, such as dynamic focussing by posing goals for the reasoning, or making (additional) assumptions during the reasoning, thus using a dynamic set of premises within the reasoning process. Also, dynamically initiated additional observations or tests to verify assumptions may be part of a reasoning process. Decisions made during the process, for example, on which reasoning goal to pursue, or which assumptions to make, are an inherent part of such a reasoning process. Such reasoning processes or their outcomes cannot be understood, justified or explained without taking into account these dynamic aspects.

The approach to the semantic formalization of the dynamics of reasoning exploited here is based on the concepts pf reasoning state, transitions and traces.

Reasoning state. A reasoning state formalizes an intermediate state of a reasoning process. The set of all reasoning states is denoted by RS.

Transition of reasoning states. A transition of reasoning states or reasoning step is an element < S, S' > of RS x RS. A *reasoning transition relation* is a set of these transitions, or a relation on RS x RS that can be used to specify the allowed transitions.

Reasoning trace. Reasoning dynamics or reasoning behaviour is the result of successive transitions from one reasoning state to another. A time-indexed sequence of reasoning states is constructed over a given timeframe (e.g. the natural numbers). Reasoning traces are sequences of reasoning states such that each pair of successive reasoning states in such a trace forms an allowed transition. A trace formalizes one specific line of reasoning. A set of reasoning traces is a declarative description of the semantics of the behaviour of a reasoning process; each reasoning trace can be seen as one of the alternatives for the behaviour. In Section 3, a language is introduced in which it is possible to express dynamic properties of reasoning traces.

The specific reasoning pattern used in this paper to illustrate the approach is 'reasoning by assumption'. This type of reasoning often occurs in practical reasoning; for example, in everyday reasoning, diagnostic reasoning based on causal knowledge, and reasoning based on natural deduction. An example of everyday reasoning by assumption is 'Suppose I do not take my umbrella with me. Then, if it starts raining at 5 pm, I will get wet, which I don't want. Therefore I'd better take my umbrella with me'. An example of diagnostic reasoning by assumption in the context of a car that won't start is: 'Suppose the battery is empty, then the lights won't work. But if I try, the lights turn out to work. Therefore the battery is not empty.' Examples of reasoning by assumption in natural deduction are as follows. Method of indirect proof: 'If I assume A, then I can derive a contradiction. Therefore I can derive not A.'. Reasoning by cases: 'If I assume A, I can derive C. If I assume B, I can also derive C. Therefore I can derive C from A or B.'.

Notice that in all of these examples, first a reasoning state is entered in which some fact is *assumed*. Next (possibly after some intermediate steps), a reasoning state is entered where *consequences* of this assumption have been *predicted*. Finally, a reasoning state is entered in which an *evaluation* has taken place; possibly in the next state the assumption is retracted, and conclusions of the whole process are added. In Section 3 and 4, this pattern is to be characterized by requirements.

3 Dynamic Properties

To specify properties on the dynamics of reasoning, the temporal trace language TTL used in [5] is adopted. This is a language in the family of languages to which situation calculus [15], event calculus [16] and fluent calculus [17] also belong.

Ontology. An ontology is a specification (in order-sorted logic) of a vocabulary. For the example reasoning pattern 'reasoning by assumption' the state ontology includes binary relations such as assumed, rejected, on sorts INFO_ELEMENT x SIGN and the relation prediction_for on INFO_ELEMENT x SIGN x INFO_ELEMENT x SIGN. Table 1 contains all the relations that will be used in this paper, as well as their explanation. The sort INFO_ELEMENT includes specific domain statements such as car_starts, lights_burn, battery_empty, sparking_plugs_problem. The sort SIGN consists of the elements pos and neg.

Table 1. State ontology for the pattern 'reasoning by assumption'

Internal concepts:	
initial_assumption(A:INFO_ELEMENT, S:SIGN)	The agent beliefs that it is most plausible to assume (A,S). Therefore, this is the agent's default assumption. For example, if it is most likely that the battery is empty, this is indicated by initial_assumption(battery_empty, pos).
assumed(A:INFO_ELEMENT, S:SIGN)	The agent currently assumes (A,S).
prediction_for(A:INFO_ELEMENT, S1:SIGN, B:INFO_ELEMENT, S2:SIGN)	The agent predicts that if (B,S2) is true, then (A,S1) should also be true.
rejected(A:INFO_ELEMENT, S:SIGN)	The agent has rejected the assumption (A,S).
alternative_for(A:INFO_ELEMENT, S1:SIGN, B:INFO_ELEMENT, S2:SIGN)	The agent beliefs that (A,S1) is a good alternative assumption in case (B,S2) is rejected.
Input and output concepts:	
To_be_observed(A:INFO_ELEMENT)	The agent starts observing whether A is true.
observation_result(A:INFO_ELEMENT, S:SIGN)	If S is pos, then the agent observes that A is true. If S is neg, then the agent observes that A is false.
External concepts:	
domain_implies(A:INFO_ELEMENT, S1:SIGN, B:INFO_ELEMENT, S2:SIGN)	Under normal circumstances, (A,S1) leads to (B,S2). For example, an empty battery usually implies that the lights do not work.
holds_in_world(A:INFO_ELEMENT, S:SIGN)	If S is pos, then A is true in the world. If S is neg, then A is false.

Reasoning state. A (reasoning) state for ontology Ont is an assignment of truth-values {true, false} to the set of ground atoms At(Ont). The set of all possible states for ontology Ont is denoted by STATES(Ont). A part of the description of an example reasoning state S is:

assumed(battery_empty, pos) : true
prediction_for(lights_ burn, neg, battery_empty, pos) : true
observation_result(lights_burn, pos) : true
rejected(battery_empty, pos) : false

The standard satisfaction relation |== between states and state properties is used: S |== p means that state property p holds in state S. For example, in the reasoning state S above it holds that S |== assumed(battery_empty, pos).

Reasoning trace. To describe dynamics, explicit reference is made to time in a formal manner. A fixed timeframe T is assumed that is linearly ordered. Depending on the application, for example, it may be dense (e.g. the real numbers) or discrete (e.g. the set of integers or natural numbers or a finite initial segment of the natural numbers). A trace γ over an ontology Ont and timeframe T is a mapping $\gamma : T \rightarrow$ STATES(Ont), i.e. a sequence of reasoning states γ_t ($t \in T$) in STATES(Ont). Please note that in each trace, the current world state is included.

Expressing dynamic properties. States of a trace can be related to state properties via the formally defined satisfaction relation |== between states and formulae. Comparable to the approach in situation calculus, the sorted predicate logic temporal trace language TTL is built on atoms such as state(γ , t) |== p, referring to traces, time and state properties. This expression denotes that state property p is true in the state of trace γ at time point t. Here |== is a predicate symbol in the language (in infix notation), comparable to the Holds-predicate in situation calculus. Temporal formulae are built using the usual logical connectives and quantification (for example, over traces, time and state properties). The set TFOR(Ont) is the set of all temporal formulae that only make use of ontology Ont. We allow additional language elements as abbreviations of formulae of the temporal trace language. The fact that this language

is formal allows for precise specification of dynamic properties. Moreover, editors can and actually have been developed to support specification of properties. Specified properties can be checked automatically against example traces to find out whether they hold.

Simulation. A simpler temporal language has been used to specify simulation models. This temporal language, the LEADSTO language [11], offers the possibility of modelling direct temporal dependencies between two state properties in successive states. This executable format is defined as follows. Let α and β be state properties of the form 'conjunction of atoms or negations of atoms', and e, f, g, h non-negative real numbers. In the LEADSTO language $\alpha \rightarrow\!\!\!\!\rightarrow_{e, f, g, h} \beta$, means:

If state property α holds for a certain time interval with duration g,
then after some delay (between e and f) state property β will hold
for a certain time interval of length h.

For a precise definition of the LEADSTO format, see [11]. A specification of dynamic properties in LEADSTO format has as advantages that it is executable and that it can easily be depicted graphically.

4 Dynamic Properties as Characterizing Requirements

Careful analysis of the informal reasoning patterns discussed in Section 2 led to the identification of dynamic properties that can serve as requirements for the capability of reasoning by assumption. In this section, a number of the most relevant of those properties are presented in both an informal and formal way. The dynamic properties identified are at three different levels of aggregation:

- **Local properties** address the step-by-step reasoning process of the agent. They represent specific transitions between states of the process: *reasoning steps*. These properties are represented in *executable* format, which means that they can be used to generate simulation traces.
- **Global properties** address the *overall* reasoning behaviour of the agent, not the step-by-step reasoning process of the agent. Some examples of global properties are presented, regarding matters as termination, correct reasoning and result production.
- **Intermediate properties** are properties at an intermediate level of aggregation, which are used for the analysis of global properties (see also Section 5).

A number of local properties are given in Section 4.1. It will be shown how they can be used in order to generate simulation traces. Next, Section 4.2 provides some global properties and Section 4.3 some intermediate properties.

4.1 Local Dynamic Properties

At the lowest level of aggregation, a number of dynamic properties have been identified for the process of reasoning by assumption. These local properties are given below (both in an informal and in formal LEADSTO notation):

LP1 (Assumption Initialization)
The first local property LP1 expresses that a first assumption is made. Here, note that initial_assumption is an agent-specific predicate, which can be varied for different cases. Formalization:
initial_assumption(A, S) $\rightarrow\!\!\!\!\rightarrow_{0,0,1,1}$ assumed(A, S)

LP2 (Prediction Effectiveness)
Local property LP2 expresses that, for each assumption that is made, all relevant predictions are generated.
Formalization:
assumed(A, S1) and domain_implies(A, S1, P, S2) $\rightarrow\!\!\!\!\rightarrow_{0,0,1,1}$ prediction_for(P, S2, A, S1)

LP3 (Observation Initiation Effectiveness)
Local property LP3 expresses that all predictions made will be observed. Formalization:
prediction_for(P, S1, A, S2) $\rightarrow\!\!\!\!\rightarrow_{0,0,1,1}$ to_be_observed(P)

LP4 (Observation Result Effectiveness)
Local property LP4 expresses that, if an observation is made, the appropriate observation result will be received. Formalization:
to_be_observed(P) and holds_in_world(P, S) $\rightarrow\!\!\!\!\rightarrow_{0,0,1,1}$ observation_result(P, S)

LP5 (Evaluation Effectiveness)
Local property LP5 expresses that, if an assumption was made and a related prediction is falsified by an observational result, then the assumption is rejected. Formalization:
assumed(A, S1) and prediction_for(P, S2, A, S1) and observation_result(P, S3) and S2≠S3 $\rightarrow\!\!\!\!\rightarrow_{0,0,1,1}$ rejected(A, S1)

LP6 (Assumption Effectiveness)
Local property LP6 expresses that, if an assumption is rejected and there is still an alternative assumption available, this will be assumed. Formalization:
assumed(A, S1) and rejected(A, S1) and alternative_for(B, S2, A, S1) and not rejected(B, S2) $\rightarrow\!\!\!\!\rightarrow_{0,0,1,1}$ assumed(B, S2)

LP7 (Assumption Persistence)
Local property LP7 expresses that assumptions persist as long as they are not rejected. Formalization:
assumed(A, S) and not rejected(A, S) $\rightarrow\!\!\!\!\rightarrow_{0,0,1,1}$ assumed(A, S)

LP8 (Rejection Persistence)
Local property LP8 expresses that rejections persist. Formalization:
rejected(A, S) $\rightarrow\!\!\!\!\rightarrow_{0,0,1,1}$ rejected(A, S)

LP9 (Observation Result Persistence)
Local property LP9 expresses that observation results persist. Formalization:
observation_result(P, S) $\rightarrow\!\!\!\!\rightarrow_{0,0,1,1}$ observation_result(P, S)

Using the software environment that is described in [11], these local dynamic properties can be used to generate simulation traces. Using such traces, the requirements engineers and system designers obtain a concrete idea of the intended flow of events over time. A number of simulation traces have been created for several

domains. An example simulation trace in the domain of car diagnosis is depicted in Fig. 1. Here, time is on the horizontal axis, and the state properties on the vertical axis. A dark box on top of the line indicates that the state property is true during that time period, and a lighter box below the line indicates that the state property is false. This figure shows the characteristic cyclic process of reasoning by assumption: making assumptions, predictions and observations for assumptions, then rejecting assumptions and creating new assumptions. As can be seen in Fig. 1, it is first observed that the car does not start. On the basis of this observation, an initial assumption is made that this is due to an empty battery. However, if this assumption turns out to be impossible (because the lights are working), this assumption is rejected. Instead, a second assumption is made (there is a sparking plugs problem), which turns out to be correct.

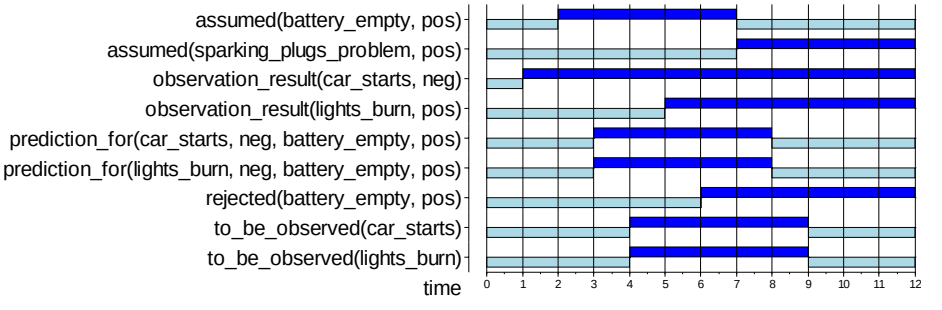

Fig. 1. Example simulation trace

4.2 Global Dynamic Properties

At the highest level of aggregation, a number of dynamic properties have been identified for the overall reasoning process. These global properties are given below (both in an informal and in formal TTL notation). Note that, in each formula, γ stands for a trace.

GP1 (Reasoning Termination)
Eventually there is a time point at which the reasoning terminates.
$\exists t{:}T \quad termination(\gamma, t)$

Here $termination(\gamma, t)$ is defined as follows:
$\forall t'{:}T \quad t' \geq t \Rightarrow state(\gamma, t) = state(\gamma, t')$.

GP2 (Correctness of Rejection)
Everything that has been rejected does not hold in the world situation.
$\forall t{:}T \; \forall A{:}INFO\_ELEMENT \; \forall S{:}SIGN$
$state(\gamma,t) \models rejected(A,S) \Rightarrow$
$state(\gamma,t) \not\models holds\_in\_world(A,S)$

GP3 (At least one not Rejected Assumption)
If the reasoning has terminated, then there is at least one assumption that has been evaluated and not rejected.

∀t:T termination(γ, t)
⇒ [∃ A: INFO_ELEMENT, ∃ S: SIGN
state(γ, t) |== assumed(A, S) ∧ state(γ, t) |=/= rejected(A, S)]

In addition, some *assumptions on the domain* can be specified:

WP1 (Static World)
If something holds in the world, it will hold for all time.
∀t:T ∀A:INFO_ELEMENT ∀S:SIGN
state(γ,t) |== holds_in_world(A,S) ⇒
[∀t':T ≥ t:T state(γ,t') |== holds_in_world(A,S)]
∀t:T ∀A:INFO_ELEMENT ∀S:SIGN
state(γ,t) |=/= holds_in_world(A,S) ⇒
[∀t':T ≥ t:T state(γ,t') |=/= holds_in_world(A,S)]

WP2 (World Consistency)
If something holds in the world, then its complement does not hold.
∀t:T ∀A:INFO_ELEMENT ∀S1,S2:SIGN
state(γ,t) |== holds_in_world(A,S1) ∧ S1 ≠ S2 ⇒
state(γ,t) |=/= holds_in_world(A,S2)

DK1 (Domain Knowledge Correctness)
The domain-specific knowledge is correct in the world.
∀t:T ∀A,B:INFO_ELEMENT ∀S1,S2:SIGN
state(γ,t) |== holds_in_world(A,S1) ∧ domain_implies(A,S1,B,S2)
⇒ state(γ,t) |== holds_in_world(B,S2)]

4.3 Intermediate Dynamic Properties

In the sections above, on the one hand, global properties for a reasoning process as a whole have been identified. On the other hand, at the lowest level of aggregation, local (executable) properties representing separate reasoning steps have been identified. It may be expected that any trace that satisfies the local properties automatically will satisfy the global properties (semantic entailment). As a form of verification, it can be proven that the local properties indeed imply the global properties. To construct a transparent proof, a number of *intermediate properties* have been identified. Examples of intermediate properties are property IP1 to IP7 shown below (both in an informal and in formal TTL notation).

IP1 (Proper Rejection Grounding)
If an assumption is rejected, then earlier on there was a prediction for it that did not match the corresponding observation result.
∀t:T ∀A:INFO_ELEMENT ∀S1:SIGN
state(γ,t) |== rejected(A,S1) ⇒
[∃t':T ≤ t:T ∃B:INFO_ELEMENT ∃S2,S3:SIGN
state(γ,t') |== prediction_for(B,S2,A,S1) ∧
state(γ,t') |== observation_result(B,S3) ∧ S2 ≠ S3]

IP2 (Prediction-Observation Discrepancy implies Assumption Incorrectness)
If a prediction does not match the corresponding observation result, then the associated assumption does not hold in the world.

∀t:T ∀A,B:INFO_ELEMENT ∀S1,S2,S3:SIGN
  state(γ,t) |== prediction_for(B,S2,A,S1) ∧
  state(γ,t) |== observation_result(B,S3) ∧ S2 ≠ S3 ⇒
    state(γ,t) |=/= holds_in_world(A,S1)

IP3 (Observation Result Correctness)
Observation results obtained from the world indeed hold in the world.
∀t:T ∀A:INFO_ELEMENT ∀S:SIGN
  state(γ,t) |== observation_result(A,S) ⇒
    state(γ,t) |== holds_in_world(A,S)

IP4 (Incorrect Prediction implies Incorrect Assumption 1)
If a prediction does not match the facts from the world, then the associated assumption does not hold either.
∀t:T ∀A,B:INFO_ELEMENT ∀S1,S2,S3:SIGN
  state(γ,t) |== prediction_for(B,S2,A,S1) ∧
  state(γ,t) |== holds_in_world(B,S3) ∧ S2 ≠ S3 ⇒
    state(γ,t) |=/= holds_in_world(A,S1)

IP5 (Observation Result Grounding)
If an observation has been obtained, then earlier on the corresponding fact held in the world.
∀t:T ∀A:INFO_ELEMENT ∀S:SIGN
  state(γ,t) |== observation_result(A,S) ⇒
    [∃t':T ≤ t:T state(γ,t') |== holds_in_world(A,S)]

IP6 (Incorrect Prediction implies Incorrect Assumption 2)
If a prediction does not hold in the world, then the associated assumption does not hold either.
∀t:T ∀A,B:INFO_ELEMENT ∀S1,S2:SIGN
  state(γ,t) |== prediction_for(B,S2,A,S1) ∧
  state(γ,t) |=/= holds_in_world(B,S2) ⇒
    state(γ,t) |=/= holds_in_world(A,S1)

IP7 (Prediction Correctness)
If a prediction is made for an assumption that holds in the world, then the prediction also holds.
∀t:T ∀A,B:INFO_ELEMENT ∀S1,S2:SIGN
  state(γ,t) |== prediction_for(B,S2,A,S1) ∧
  state(γ,t) |== holds_in_world(A,S1) ⇒
    state(γ,t) |== holds_in_world(B,S2)

5 Relationships Between Dynamic Properties

A number of logical relationships have been the identified between properties at different aggregation levels. An overview of all identified logical relationships relevant for GP2 is depicted as an AND-tree in Fig. 2. Here the grey ovals indicate that the so-called *grounding* variant of the property is used. Grounding variants make a specification of local properties more complete by stating that there is no other means to produce certain behaviour. For example, the grounding variant of LP2 can be specified as follows (in TTL notation):

LP2G Prediction effectiveness groundedness
Each prediction is related (via domain knowledge) to an earlier made assumption.
∀t:T ∀A,B:INFO_ELEMENT ∀S1,S2:SIGN
state(γ,t) |== prediction_for(B,S2,A,S1) ⇒
[∃t':T ≤ t:T state(γ,t') |== assumed(A,S1) ∧
domain_implies(A,S1,B,S2)]

This property expresses that predictions made always have to be preceded by a state in which the assumption was made, and the domain knowledge implies the prediction.

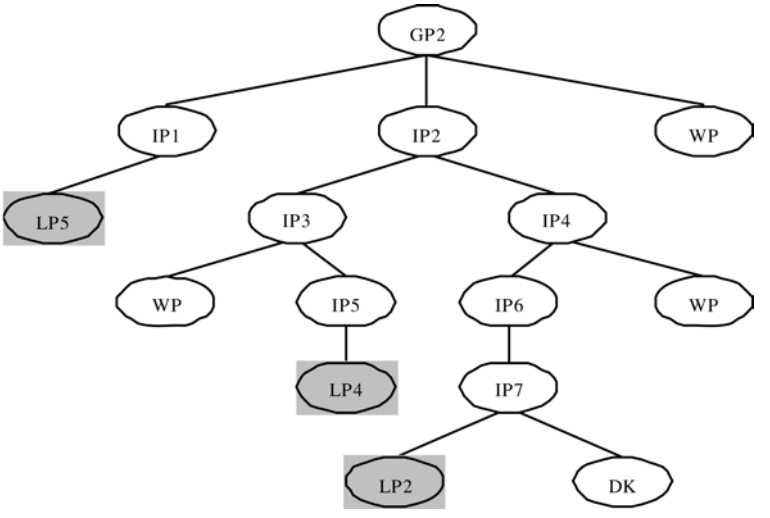

Fig. 2. AND-tree of dynamic properties

The relationships depicted in Fig. 2 should be interpreted as semantic entailment relationships. For example, the relationship at the highest level expresses that the implication IP1 & IP2 & WP1 => GP2 holds. A sketch of the proof for this implication is as follows.

> *Suppose IP1 holds. This means that, if an assumption is rejected at time t, then at a certain time point in the past (say t') there was a prediction for it that did not match the corresponding observation result. According to IP2, at the very same time point (t') the assumption for which the prediction was made did not hold in the world. Since the world is static (WP1), this assumption still does not hold at time point t. We may thus conclude that, if something is rejected at a certain time point, it does not hold in the world.*

Logical relationships between dynamic properties can be very useful in the analysis of empirical reasoning processes. For example, if a given person makes an incorrect rejection (i.e. property GP2 is not satisfied by the reasoning trace), then by a refutation process it can be concluded that either property IP1, property IP2, or property WP1 fails (or a combination of them). If, after checking these properties, it turns out that IP1 does not hold, then this must be the case because LP5G does not

hold. Thus, by this example refutation analysis, it can be concluded that the cause of the unsatisfactory reasoning process can be found in LP5G. For more information about the analysis of human reasoning processes, see [12].

6 Verification

In addition to the simulation software described in Section 4, a special tool has been developed that takes a formally specified property and a set of traces as input, and verifies whether the property holds for the traces.

Using this checker tool, dynamic properties (of all levels) can be checked automatically against traces, irrespective of who/what produced those traces: humans, simulators or an implemented (prototype) system. A large number of such checks have indeed been performed for several case studies in reasoning by assumption. Table 2 presents an overview of all combinations of checks and their results. A '+' indicates that all properties were satisfied for the traces, a '+/-' indicates that some of the properties were satisfied.

Table 2. Overview of the different verification results

	Human Traces (Taken from [12])	**Simulation Traces** (This paper)	**Prototype Traces** (Taken from [13])
Local Properties	+/-	+	+
Intermediate Properties	+/-	+	+
Global Properties	+/-	+	+

As can be seen in Table 2, three types of traces were considered. First, the dynamic properties have been checked against human traces in reasoning experiments. It turned out that some of the properties were satisfied by all human traces, whereas some other properties sometimes failed. This implies that some properties are indeed characteristic for the pattern 'reasoning by assumption', whereas some other properties can be used to discriminate between different approaches to the reasoning. For example, human reasoners sometimes skip a step; therefore LP2 does not always hold. More details of these checks can be found in [12].

Second, the dynamic properties have been checked against simulation traces such as the one presented in Section 4.1 of this paper. As shown in Table 2, all properties eventually were satisfied for all traces. Note that this was initially not the case: in some cases, small errors were made during the formalization of the properties. Checking the properties against simulation traces turned out to be useful to localize such errors and thereby debug the formal dynamic properties.

Finally, all dynamic properties have been verified against traces generated by a prototype of a software agent performing reasoning by assumption [13]. This agent was designed on the basis of the component-based design method DESIRE, cf. [14]. Also for these traces eventually all dynamic properties turned out to hold.

To conclude, all automated checks described above have played an important role in the requirements analysis of reasoning capabilities of software agents, since they

permitted the results of the requirements elicitation and specification phase to be formally verified and improved.

Note that, although the dynamic properties shown in the previous sections are mainly aimed at functional requirements, in principle the approach based on TTL allows one to verify non-functional requirements as well. Examples of non-functional requirements are efficiency, reliability and portability of the system [18]. Despite the fact that these types of requirements are generally difficult to formalize, some initial steps have been made towards their formalization in TTL [19]. In that paper, it is suggested that the efficiency of a system can be measured, for example, by counting the amount of components that need to be activated in order to be successful. This property is formalized in TTL as follows:

efficiency(γ:TRACE) ≡
  successfulness(γ) ∧
  ∃i :INTEGER component_activations(γ, i) ∧ i = shortest_path

Here, it is assumed that the length of the shortest path is known for the particular example being checked. To enable a definition of the number of activations of a component, first the activation of one component is defined, including its interval:

has_activation_interval(γ:TRACE, c:COMPONENT, tb:TIME, te:TIME) ≡
  tb < te ∧ state(γ,te) |≠ activated(c) ∧
  [∀t tb≤t<te ⇒ state(γ,t) |= activated(c)] ∧
  ∃t1<tb [∀t2 t1≤t2<tb ⇒ state(γ,t2) |≠ activated(c)]

An example of a definition for a trace with one component activation is shown below.

component_activations(γ:TRACE, 1) ≡
  ∃c:COMPONENT, tb:TIME, te:TIME
  has_activation_interval(γ, c:COMPONENT, tb:TIME, te:TIME) ∧
  [∀c2:COMPONENT, tb2:TIME, te2:TIME
  [has_activation_interval(γ, c2:COMPONENT, tb2:TIME, te2:TIME) ⇒
    c = c2 ∧ tb = tb2 ∧ te = te2]]

Another way to describe efficiency is by considering the amount of computation time the approach needs to generate a solution.

7 Discussion

In the literature, software engineering aspects of reasoning capabilities of intelligent agents have not been addressed well. Some literature is available on formal semantics of the dynamics of non-monotonic reasoning processes; for an overview see [9]. However, these approaches focus on formal foundation and are far from the more practical software engineering aspects of actual agent system development.

In this paper, it is shown how, during an agent development process, a requirements analysis can be incorporated. The desired functionality of the agent's reasoning capabilities can be identified (for example, in cooperation with stakeholders), using temporal specifications of scenarios and requirements specified in the form of (required) traces and dynamic properties. This paper shows, for the example reasoning pattern 'reasoning by assumption', how relevant dynamic properties can be identified as requirements for the agent's reasoning behaviour, expressed in a temporal language, and

verified and validated. Thus a set of requirements is obtained that is reusable in other agent development processes. The main reason for the reusability of these requirements is the fact that, within the presented dynamic properties, generic and domain-specific concepts can be treated separately (compositionality of knowledge). For example, in global property DK1, the domain-specific sort INFO_ELEMENT and the relation domain_implies can be filled in for any specific case. This allows the software engineer to reuse the presented requirements in any given domain, as long as it involves reasoning by assumption. In fact, the approach has already been applied to several different examples: whereas the main reasoning problem addressed in the current paper is about 'car diagnosis', other reasoning problems have been addressed in the past, among which the 'wise person's puzzle' [13] and the game of 'Mastermind' [12].

The language TTL used here allows for precise specification of the requirements for an agent's reasoning behaviour, covering both qualitative and quantitative aspects of states and their temporal relations. Moreover, software tools have been developed to (1) support specification of (executable) dynamic properties, and (2) automatically check specified dynamic properties against example traces to find out whether the properties hold for the traces. This provides a useful supporting software environment to evaluate reasoning scenarios both in terms of simulated and prototype traces (in the context of prototyping) and empirical traces (in the context of requirements elicitation and validation in co-operation with stakeholders). In the paper, it is shown how this software environment can be used to automatically check the dynamic properties during a requirements analysis process. Note that it is not claimed that TTL is the only language appropriate for this. For example, most of the properties encountered could as well have been expressed in a variant of linear time temporal logic. The language is only used as a vehicle; the contribution of the paper is in the method of application of requirements analysis to an agent's reasoning capability, and the reusable results obtained by that method.

For an elaborate description about the role that the current approach may take in requirements engineering, the reader is referred to [20]. In that paper, it is shown in detail how dynamic properties can be used to specify (both functional and non-functional) requirements of agent systems. Moreover, it is shown how these requirements may be refined and fulfilled according to the Generic Design Model (GDM) by Brazier *et al.* [21]. However, GDM is just one possible approach for Agent-Oriented Software Engineering. Recently, several other architectures have been proposed, for example, Tropos [22], KAOS [23] or GBRAM [24]. In future work, the possibilities may be explored to incorporate the approach based on dynamic properties presented here within such architectures. These possibilities are promising, especially for architectures that provide a specific language for formalization of requirements (KAOS for example uses a real-time temporal logic to specify requirements in terms of goals, constraints and objects).

References

1. Dardenne, A., Lamsweerde, A. van, and Fickas, S.: Goal-directed Requirements Acquisition. *Science in Computer Programming*, vol. 20 (1993) 3-50
2. Kontonya, G., and Sommerville, I.: *Requirements Engineering: Processes and Techniques*. John Wiley and Sons, New York (1998)

3. Sommerville, I., and Sawyer P.: *Requirements Engineering: a good practice guide*. John Wiley & Sons, Chicester, England (1997)
4. Dubois, E., Du Bois, P., and Zeippen, J.M.: A Formal Requirements Engineering Method for Real-Time, Concurrent, and Distributed Systems. In: *Proceedings of the Real-Time Systems Conference, RTS'95* (1995)
5. Herlea, D.E., Jonker, C.M., Treur, J., and Wijngaards, N.J.E.: Specification of Behavioural Requirements within Compositional Multi-Agent System Design. In: F.J. Garijo, M. Boman (eds.), *Multi-Agent System Engineering, Proc. of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'99*. LNAI, vol. 1647, Springer Verlag (1999) 8-27
6. Treur, J.: Semantic Formalisation of Interactive Reasoning Functionality. *International Journal of Intelligent Systems*, vol. 17 (2002) 645-686
7. Leemans, N.E.M., Treur, J., and Willems, M.: A Semantical Perspective on Verification of Knowledge. *Data and Knowledge Engineering*, vol. 40 (2002) 33-70
8. Engelfriet, J., and Treur, J.: Temporal Theories of Reasoning. *Journal of Applied Non-Classical Logics*, 5 (1995) 239-261
9. Meyer, J.-J., Ch., and Treur, J. (eds.): *Dynamics and Management of Reasoning Processes*. Series in Defeasible Reasoning and Uncertainty Management Systems (D. Gabbay, Ph. Smets, series eds.), Kluwer Acad. Publishers (2001)
10. Barringer, H., Fisher, M., Gabbay, D., Owens, R., and Reynolds, M.: *The Imperative Future: Principles of Executable Temporal Logic*, Research Studies Press Ltd. and John Wiley & Sons (1996)
11. Bosse, T., Jonker, C.M., Meij, L. van der, and Treur, J.: LEADSTO: a Language and Environment for Analysis of Dynamics by SimulaTiOn. In: Eymann, T. *et al.* (eds.), *Proceedings of the 3rd German Conference on Multi-Agent System Technologies, MATES'05*. Lecture Notes in AI, vol. 3550, Springer Verlag (2005) 165-178
12. Bosse, T., Jonker, C.M., and Treur, J.: Formalization and Analysis of Reasoning by Assumption. *Cognitive Science Journal*, vol. 30, issue 1 (2006) 147-180
13. Jonker, C.M., and Treur, J.: Modelling the Dynamics of Reasoning Processes: Reasoning by Assumption. *Cognitive Systems Research Journal*, vol. 4 (2003) 119-136
14. Brazier, F.M.T., Jonker, C.M., and Treur, J.: Principles of Component-Based Design of Intelligent Agents. *Data and Knowledge Engineering*, vol. 41 (2002) 1-28
15. Reiter, R.: Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. MIT Press (2001)
16. Kowalski, R., and Sergot, M.: A logic-based calculus of events. *New Generation Computing*, vol. 4 (1986) 67-95
17. Hölldobler, S., and Thielscher, M.: A new deductive approach to planning. *New Generation Computing*, vol. 8 (1990) 225-244
18. Davis, A.M.: *Software Requirements: Objects, Functions, and States*. Prentice Hall (1993)
19. Bosse, T., Hoogendoorn, M., and Treur, J.: Automated Evaluation of Coordination Approaches. In: *Proceedings of the Eighth International Conference on Coordination Models and Languages, Coordination'06*. Lecture Notes in Computer Science, vol. 4038. Springer Verlag (2006) 44-62
20. Bosse, T., Jonker, C.M., and Treur, J.: Analysis of Design Process Dynamics. In: R. Lopez de Mantaras, L. Saitta (eds.), *Proceedings of the 16th European Conference on Artificial Intelligence, ECAI'04*, IOS Press (2004) 293-297
21. Brazier F.M.T., Langen P.H.G. van, Treur J.: Strategic knowledge in design: a compositional approach. In: K. Hori (ed.), *Knowledge-Based Systems*. Special Issue on Strategic Knowledge and Concept Formation, vol. 11, issue 7-8 (1998) 405-416

22. Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., and Perini, A.: Tropos: An Agent-Oriented Software Development Methodology. *Journal of Autonomous Agent and Multi-Agent Systems*, vol. 8 (2004) 203-236
23. Darimont, R., Delor, E., Massonet, P., and van Lamsweerde, A.: GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering, *Proc. ICSE'98 - 20th International Conference on Software Engineering*, Kyoto, vol. 2 (1998) 58-62
24. Antón, A.I.: Goal-based Requirements Analysis, *Proc. of the International Conference on Requirements Engineering (ICRE'96)*, IEEE Computer Soc. Press, Colorado Springs, Colorado, USA (1996) 136-144

On the Cost of Agent-Awareness for Negotiation Services

Andrea Giovannucci and Juan A. Rodríguez-Aguilar

Artificial Intelligence Research Institute, IIIA
Spanish Council for Scientific Research, CSIC
08193 Bellaterra, Barcelona, Spain
{andrea, jar}@iiia.csic.es

Abstract. Significant advances in the development of agent technology have spurred the development of agent-oriented information systems (AOIS). Nonetheless, accounts on the benefits and shortcomings of state-of-the-art agent technology when employed for the deployment of AOIS for electronic commerce are scant. The purpose of this work is to report on a case study that attempts to shedd some light on this matter.

1 Introduction

While a significant number of agent-based applications for electronic commerce have been presented to the agent community during the last years, little attention has been devoted to analysing the practical benefits and shortcomings of agent technology when applied to such domain. To the best of our knowledge, little effort has been devoted to study the applicability of state-of-the-art agent technology to develop actual-world e-commerce applications. In particular, we believe that it is necessary to assess the computational cost added by agent technology in this type of applications so that we can diagnose the improvements required by state-of-the-art agent technology.

For this purpose, we report on a case study that is intended to shed some light on this matter. We depart from *iBundler* (fully described in [1]), an agent-aware negotiation service for combinatorial negotiations designed to be employed as: (1) an open agent platform within the Agentcities.RDT[1] (http://www.agentcities.org/EURTD) project that could be discovered, communicate and offer services to any FIPA compliant agent (http://www.fipa.org); (2) an agent façade to *Quotes*[2], a commercial negotiation tool, to allow for the participation of third-party business agents in actual-world procurement events. In both cases, our aim has been to study the computational cost of agent awareness for the *iBundler* negotiation service so that its users are aware of the type of negotiation scenarios that *iBundler* can acceptably handle when buying and providing agents are involved. This exercise has also included the determination of those general or domain-dependent measures that can help reduce the cost of the service.

At this aim, we have measured the performance in time and memory of *iBundler* through a wide range of artificially generated negotiation scenarios. For each scenario, we sampled at several stages both the time and memory that *iBundler* employed to

[1] The Agentcities.RDT project's objectives were to create an on-line, distributed test-bed to explore and validate the potential of agent technology for future dynamic service environments.

M. Kolp et al. (Eds.): AOIS 2005, LNAI 3529, pp. 64–78, 2006.

handle it. We have interestingly observed that the management of ontologies is a rather delicate issue that actually causes a significant overload. Furthermore, we have also observed that the design of highly expressive, compact bidding languages can definitely help cut down the computational cost for any agent-aware negotiation service considering combinatorial scenarios.

The paper is organized as follows. Firstly, section 2 briefly reviews the literature concerning scalability and applicability of agent technology. Section 3 succinctly introduces *iBundler*. Section 4 deals with the description of the evaluation scenarios arranged to evaluate *iBundler*. In Section 5 we present and thoroughly discuss the test results. Finally, Section 6 discusses some conclusions deriving from the results' analysis.

2 Related Work

The applicability analysis of agent technology in the literature primarily focusses on scalability issues as robustness, system performance with large populations of agents and ontology engineering. Brazier emph et al. [3] address the problem of scalability in naming services and location services. In addition, they analyse the concept of scalability in multi-agent systems (MAS) and discuss scalability for many existing multi-agent frameworks. Deters [4] studies the problems derived from large number of agents running in an MAS: agent resource consumption, the exchange of great number of messages, identifying agent hosting and message routing as bottlenecks. Furthermore, he performs some scalability experiments. An important result in [4] is that the main deficiencies of JESS (http://herzberg.ca.sandia.gov/jess/) derive from serialization processes. Kahn investigates how timing of sequential agent registration and lookup varies as the total number of registered agents increases in COABS [5]. The works in [6] and [7] analyse robustness and fault tolerance, whereas [8] exemplifies ad hoc, domain-dependent agent technology scaling techniques. On the other hand, the literature on ontology scalability focusses on three major issues: the size of ontology contents, the complexity of ontology construction and knowledge reusability ([9], [10]). In particular, Jarrar states that experience shows that "unscalable solutions emerging from academic research often fails at the industrial level" [9].

Thus, we believe that it is an urgent necessity to report on practical deployments of actual-world agent-based applications in order to: (1) progressively derive best methodological practices; and (2) assess the improvements required by state-of-the-art agent technologies to be adopted at an industry level, particularly since much of the research effort on agent technology does not consider the application of widely employed agent frameworks and programming tools to real-world problems.

We consider *iBundler* as representative of the main trends on the state-of-the-art agent programming tools and platforms, firstly, because it is based on the FIPA specification standard, probably the most widely adopted by the agent community[2]. Secondly, the considerations emerging from the experiments derived in this paper are related to

[2] OGM (www.ogm.org) is another standardization effort based on CORBA IDL interface. This solution is efficient for agent migration and client-server applications, but less suitable than FIPA-compliant platforms for peer-to-peer applications. For an interesting comparison, refer to [11].

the FIPA nature of the agent platform, not to a particular JADE implementation. Thus, the results in Section 5 are not limited to the JADE framework, being valid for all the FIPA-compliant agent frameworks.

3 *iBundler* An Agent-Aware Negotiation Service

Consider the problem faced by a buying agent when negotiating with providing agents. In a negotiation event involving multiple, highly customisable goods, buying agents need to express relations and constraints between attributes of different items. Moreover, it is common practice to buy different quantities of the very same product from different providing agents, either for safety reasons or because offer aggregation is needed to cope with high-volume demands. This introduces the need to express business constraints on providing agents and the contracts they may have assigned. Not forgetting the provider side, providing agents may also wish to impose constraints or conditions over their offers. These may be only valid if certain configurable attributes (e.g. quantity, delivery days) fall within some intervals, or assembly and packing constraints need to be considered. Once a buying agent collects all offers, he/she is faced with the burden of determining the winning offers. It would be desirable to relieve buying agents from solving such a problem. *iBundler* is an agent-aware decision support service that makes headway in this direction by acting as a combinatorial negotiation solver (solving the winner determination problem) for both multi-item, multi-unit negotiations and auctions. Thus, the service can be employed by both buying agents and auctioneers in combinatorial negotiations and combinatorial reverse auctions [12] respectively. To the best of our knowledge, *iBundler* represents the first agent-aware service for multi-item negotiations, since agent services have mostly focussed on infrastructure issues related to negotiation protocols and ontologies.

The *iBundler* service has been implemented as an agency composed of agents that cooperatively interact to offer a negotiation support service. A fundamental aspect of *iBundler* is that it was not only intended as a stand-alone agent-aware service. *iBundler* was also designed to become the agent façade of the commercial sourcing tool *Quotes* [2] with the aim of providing a higher level of automation to external parties. In this manner, the negotiations run through *Quotes* allow for the participation of both human and software buyers and providers. However, while human buyers and providers negotiate via web-based interfaces, buying and providing agents owned by third parties can also negotiate through the service whenever they incorporate protocols and the ontology required by *iBundler*. In this work, we do not address security issues, such as buyers and providers trusting a central server. It could be considered as a next step in the deployment of an actual-world negotiation service.

Fig. 1 depicts the components of the *iBundler* agency (along with the fundamental connections of buying and providing agents with the service):

[Logger agent]. It manages the access to the *iBundler* agency from outside.
[Manager agent]. Agent devoted to providing the solution of the problem of choosing the set of bids that best matches a user's requirements. There exists a single Manager agent per user (buyer or auctioneer), created by the Logger agent, offering the following services: brokering service to forward buyers requirements (RFQs) to selected providers

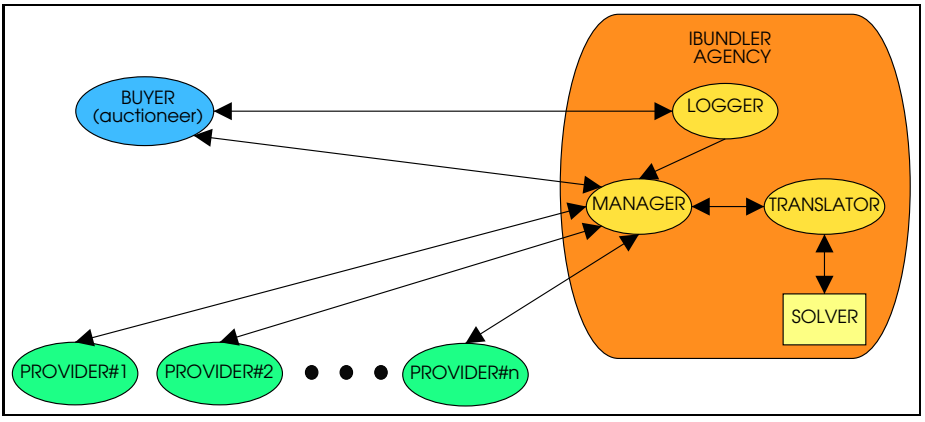

Fig. 1. Architecture of the *iBundler* Agency

capable of multi-agenting them; collection of bids; winner determination in a combinatorial negotiation/auction; and award of contracts on behalf of buyers. Furthermore, the manager agent is also responsible for bundling each RFQ and its bids into a negotiation problem in FIPA-compliant format to be conveyed to the Translator agent; and to extract the solution to the negotiation problem handled back by the Translator agent.
[Translator agent]. It creates a representation of the negotiation problem in a format understandable by the Solver departing from the FIPA-compliant description received from the Manager. It also translates the solution returned by the Solver into an object of the ontology employed by user agents.
[Solver component]. The *iBundler* component itself extended with the offering of a language for expressing offers, constraints and requirements. The specification is parsed into a Mixed Integer Programming (MIP) formulation and solved using available MIP solvers (a version using ILOG CPLEX and another version using using a Java MIP modeller that integrates the GNU (www.gnu.org) Programming Kit GLPK). The Solver component is complete in the sense that, if an optimal solution exists, it will find it. If the problem has a set of Pareto-optimal, equivalent solutions, the solver component will return only one solution, depending on the underlying branch-and-bound algorithm [13].

Our design manages to separate concerns among the three members of the agency. On the one hand, the Manager is strictly devoted to coordination. It represents the façade of the service. In addition, since every negotiation requested by a buyer makes the agency create an instance of the *Manager*, the service can cope with asynchronous and multiple accesses to the service. The Translator agent is in charge of relieving both Managers and Solver from the burden of translating FIPA-compliant specifications into the language required by Solver. Notice that the fact of having only one Translator agent represents a bottleneck in the overall process when many buyers access the service concurrently. Such a limitation could be overcome by creating multiple instances of Translator Agents and Solvers on different machines. However, in this work, we focussed on the service performances in managing large negotiation scenarios, not on multiple concurrent accesses to the service. We leave such issue as a possible future development.

Fig. 2 depicts the interaction protocol involved in the interplay of buyers and provides with *iBundler*. It is expressed in AUML (Agent Unified Modelling Language) [14] following the FIPA interaction protocol library specification compiled in [15]. Observe that the specification in Fig. 2 involves four roles, namely buyer, manager, translator and provider. Whereas multiple agents can act as providers, the remaining roles can each be uniquely adopted by a single agent. Notice too that the *iBundler* interaction protocol is composed of several interleaved interaction protocols:

[IP-RFQ]. Held between a buyer and the manager agent created by the Logger agent after registration. The buyer delivers an RFQ to his manager agent requesting the optimal set of offers from the available providers. In the case when it is not possible to obtain a solution to the problem, the received response is an empty bid set.

[IP-CFP]. Prior to delivering the optimal set of offers, the manager interacts with the available providers to request their offers under the rules of this CFP interaction protocol. If no offers are received, the manager refuses to deliver the optimal set of offers in the context of the IP-RFQ interaction protocol. Otherwise, the manager agrees on providing the service and proceeds ahead by starting an instance of the IP-Request-Solution interaction protocol. The protocol completes with the notification of contract awards to selected providers according to the buyer's decision. In the case in which no optimal solution could be found, the buyer is sent an empty bid set and the IP-CFP protocol is ended communicating a Reject-Proposal to each provider involved. Notice that the manager mediates between buyer and providers.

[IP-Request Solution]. This interaction protocol held between the manager and the translator agent within the *iBundler* agency is aimed at calculating the optimal set of offers considering the offers submitted by providers, along with the buyer's requirements and constraints. The result delivered by the translator is further conveyed by the manager to the buyer in the context of the interleaved IP-RFQ interaction protocol.

[IP-AWARD]. At the end of the IP-RFQ interaction protocol the buyer obtains the optimal set of offers. He/she may request also to receive all offers. Thereafter, if the buyer received a non-empty optimal set of offers ($k > 0$ in Fig. 2), the buyer initiates the IP-AWARD interaction protocol in order to request the manager to award contracts to selected providers. Observe that the contract award distribution is autonomously composed by the buyer, and thus the buyer may decide to either ignore or alter the optimal set.

iBundler's ontology is founded on the following core concepts: *RFQ*, *ProviderResponse*, *Problem*, and *Solution*. As an example, Fig. 3 depicts — as shown by the Ontoviz Protégé plug-in (http://protege.stanford.edu) — the *Problem* ontological concept. The RFQ concept is employed by buying agents to express their requests for bids (via a request in IP-RFQ). An *RFQ* is composed of a sequence of *Request* concepts, one per requested item, along with the buyer's business rules expressed as constraints. On the provider side, providers express their offers in terms of the *ProviderResponse* concept (via a proposal in IP-CFP), which in turn is composed of several elements: a list of *Bid* concepts (each Bid allows to express a bid per either a single requested item or a bundle of items) along with constraints on the production/servicing capabilities of the bidding provider (*Capacity* concept) and constraints on bundles of bids formulated with the *BidConstraint* concept.

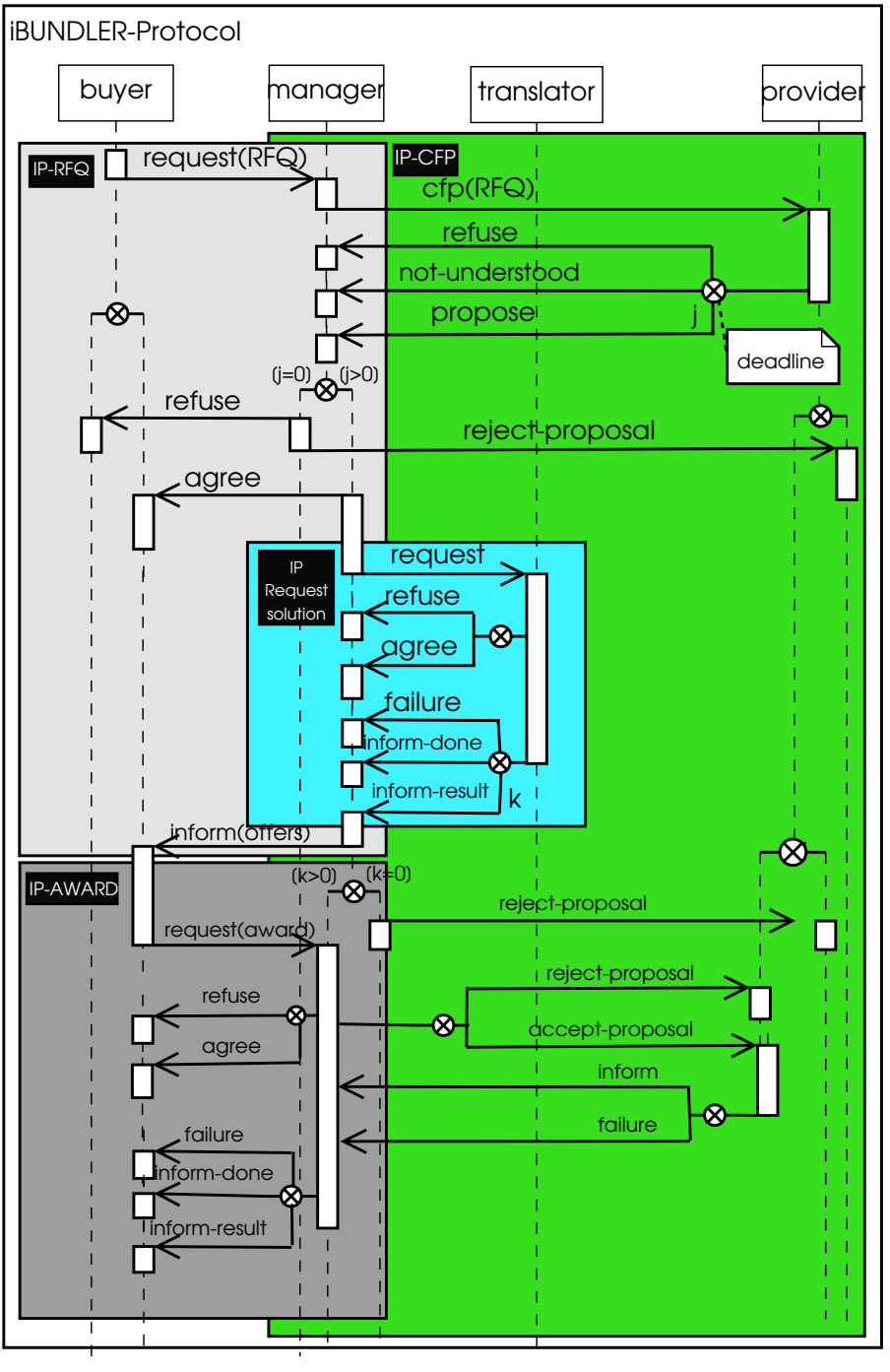

Fig. 2. *iBundler* Interaction Protocol

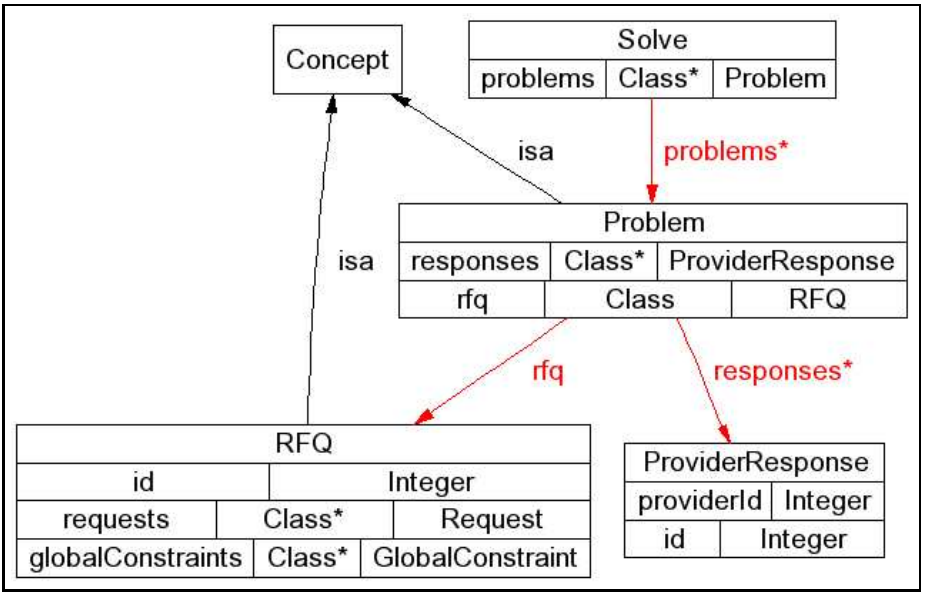

Fig. 3. Problem concept

Once the manager agent collects all offers submitted by providers, he/she wraps up the *RFQ* concept as received from the buyer along with the offers as *ProviderResponse* concepts to compose the negotiation problem to be solved by the Solver component (via a request in IP-Request-Solution). Finally, the solution produced by the Solver component is transformed by the translator agent into a *Solution* concept, which is handed over to the manager (via an inform-result in IP-Request-Solution). The *Solution* concept contains the specification of the optimal set of offers calculated by Solver. Thus *Solution* contains a list of *SolutionPerProvider* concepts, each one containing the bids selected in the optimal bid set per provider, as a list of *BidSolution* concepts, along with the provider's agent identifier, as an *AID* concept. Each *BidSolution* in turn is composed of a list of *BidItemFixed* concepts containing the number of units selected per bid along with the bid's total cost.

4 Evaluation Scenario

In this section, we detail the way we conducted our evaluation. Firstly, we describe how to generate artificial negotiation scenarios for testing purposes. Next, we detail the different stages considered through our evaluation process.

4.1 Artificial Negotiation Scenarios

In order to evaluate the agent service performance, the times needed by *iBundler* to receive an RFQ from a *Buyer* agent and to collect the different bids from providers is

considered to be of no interest because they depend on some uncontrolled variables (e.g. the time needed by providers to send their bids and the network delay). Thus, our evaluation starts from the moment at which all the required data (RFQ and bids) are available to the *Manager* agent. We tried to simulate such an ideal situation generating multiple datasets in separate files, each one standing for a different input negotiation problem composed of FIPA messages and containing both an RFQ and the bids received as a response to this. In this way, we can use the file stream as if it was the incoming message stream and perform all the subsequent message manipulation as if the message had been received from a socket.

Another important consideration has to do with the way we sampled time and memory. We established checkpoints through the process carried out by *iBundler* when solving a negotiation problem. Such checkpoints partition the process into several stages. We observed time and memory at the beginning and at the end of these stages.

In order to automate the testing, it was necessary to develop a generator of artificial negotiation scenarios involving multiple units of multiple items. The generator is fed with mean and variance values for the following parameters: *number of providers* participating in the negotiation; *number of bids per provider* (number of bids each provider sends to the *Manager* agent); *number of RFQ items* (number of items to be negotiated by the *Buyer* agent); *number of items per bid* (number of items within each bid sent by a provider); *number of units per item per bid*; and *bid cost per item*. In this first experimental scenario, we did not generate either inter-item or intra-item constraints.

The generator starts by randomly creating a set of winning combinatorial offers. After that, it generates the rest of bids for the negotiation scenario employing normal distributions based on the values set for the parameters above. Thus, in some sense, the negotiation scenario can be regarded as a set of winning combinatorial bids surrounded by noisy bids (far less competitive bids). Notice that the generator directly outputs the RFQ and bids composing an artificial negotiation scenario in FIPA format. In this manner, both RFQ and bids can be directly fed into *iBundler* as buyers' and providers' agent messages.

We have analysed the performance of *iBundler* through a large variety of negotiation scenarios artificially generated by differently setting the parameters above. The data representing each negotiation scenario are saved onto a file, named by a string of type A.B.C.D, where A stands for the number of providers, B stands for the number of bids per provider, C stands for the number of RFQ items and D stands for the number of items per bid. For instance, *250.20.100.20* represents the name of a dataset generated for 250 providers, 20 bids per provider, 100 RFQ items and 20 items per bid.

The artificial negotiation scenarios we have generated and tested result from all the possible combinations of the following values:

Number of providers: 25, 50, 75, 100
Number of bids per provider: 5, 10, 15, 20
Number of RFQ items: 5, 10, 15, 20
Number of items per bid: 5, 10, 25, 50

4.2 Evaluation Stages

In order to introduce the evaluation stages that we considered, it is necessary firstly to understand how JADE manipulates messages and ontological objects. In particular, we summarize the process of sending and receiving messages (for a complete description refer to the JADE documentation). Fig. 4 graphically summarizes the activities involved in sending and receiving messages. In the figure, the squared boxes represent data, whereas the rounded boxes represent processes.

JADE agents receive messages as serialized objects in string format. JADE decodes the string into a Java class, the *ACLMessage* JADE class (which represents a FIPA ACL Message). One of these class fields is the *content* field, which usually contains either the action to be performed or the result of a performed action. Next, JADE extracts the content of the message. The content is once more a string, on which JADE needs to perform an ontology check to decode it. As a result, a Java object representing the ontological object is built upon the *content* field, guaranteeing that the ontological structure is not violated.

As to the dual case, i.e. when a JADE agent sends a message, the process works the other way around. JADE encodes the ontological object representing the communication content into a string that sets the *content* field of the *ACLMessage* class. During this process JADE verifies that the message content matches perfectly with an ontology object. Once the *content* field is set, the agent sends the message: the ACLMessage class is decoded into a string that is sent through a socket.

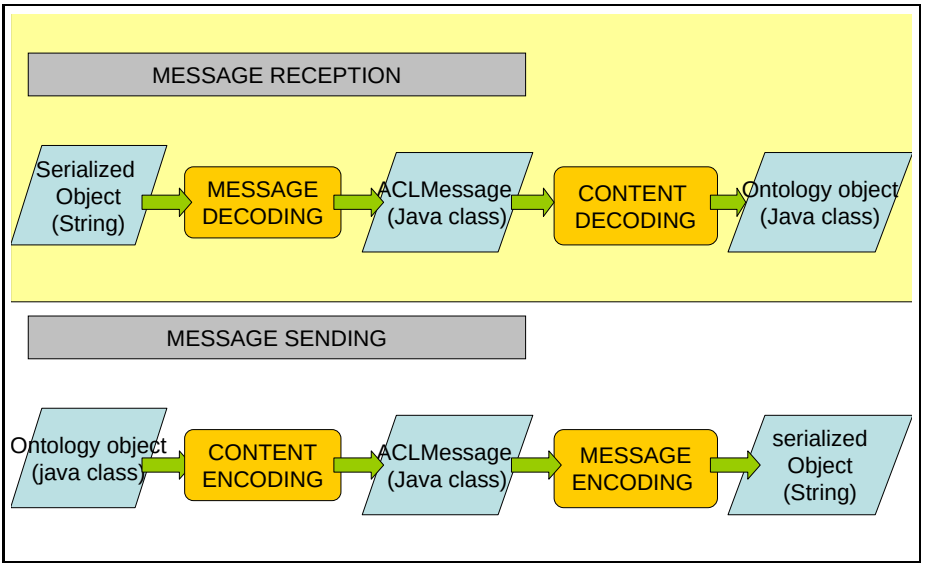

Fig. 4. Message life cycle in JADE

Considering the process above, we sampled both the time and memory usage through the following stages of the *iBundler*'s solving process:

Δ**t1**: JADE decodes all the FIPA messages contained in the data set file containing the input negotiation problem, converting them into instances of the *ACLMessage* Java class. Δ**t2**: the *Manager* agent composes the problem by creating an instance of the *Problem* Java ontology class and setting its fields after merging the RFQ and the collected bids. Δ**t3**: the ACLMessage to be sent to the *Translator Agent* is filled with the Java class representing the *Problem* ontology class. At this stage, an ontology check occurs.
Δ**t4**: the above-mentioned ACLMessage is now encoded by the *Manager* agent, and subsequently sent to the *Translator* agent through a socket. Once received, the *Translator* agent decodes it into an *ACLMessage* class.
Δ**t5**: the *Translator* agent extracts from the received message the *Problem* ontology class containing the *RFQ* and all the collected *Bids*. Another ontology check occurs.
Δ**t6**: this stage is devoted to the transformation of the *Problem* ontology class into a matrix-based format to be processed by the *Solver* component.
Δ**t7**: at this stage the *Solver* component solves the MIP problem using ILOG CPLEX.
Δ**t8**: the output generated by *Solver* in matrix-based format is decoded by the *Translator* agent into the *Solution* ontology class.
Δ**t9**: the *Translator* agent fills the response message with the *Solution* ontology class, encodes the corresponding *ACLMessage* class and sends it. Then, the *Manager* agent decodes the message upon reception.
Δ**t10**: the *Manager* agent extracts the *Solution* concept from the received *ACLMessage* with a last ontology check.
Δ**t11**: the solution is decomposed into different parts, one per provider owning an awarded bid.
Δ**t12**: the solution containing the set of winning offers is sent from the *Manager* agent to the *Buyer* agent. Note that this object is small with respect to the original problem since it only contains the winning bids.

5 Evaluation

In this section, we give a quantitative account of the tests we run. Firstly, in Section 5.1, we analyse time performance and, secondly, in Section 5.2 the memory usage for all the evaluation stages described above. In order to run our tests we employed the following technology: a PC with a Pentium IV processor, 3.1 Ghz, 1 Gbyte RAM running a Linux Debian (kernel v.2.6) operating system (http://www.debian.org); Java SDK 1.4.2.04 (http://java.sun.com); JADE v2.6; and ILOG CPLEX 9.0 (http://www.ilog.com).

5.1 Time Performance

Next, we show the variation in time performance per stage by varying the different degrees of freedom available to create an artificial negotiation scenario. In particular, we consider the following types of negotiation scenarios:

100.20.100.X: the number of items contained in a single bid varies (where X takes on the 5,10,25, and 50 values).

100.X.100.50: the number of bids each provider sends varies (where X takes on the 5,10,15, and 20 values).
X.20.100.50: the number of providers varies (where X takes on the 25,50,75, and 100 values).

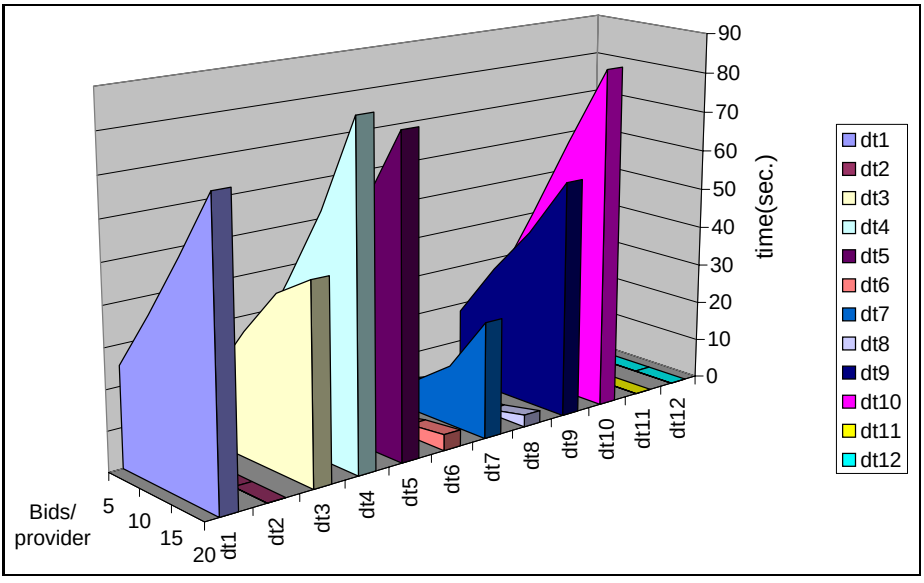

Fig. 5. Time measures when varying the number of bids per provider

Fig. 5 depicts the time spent in each of the described stages, considering different number of bids per provider. We experimented with similar trends varying the number of items and the number of providers[3]. These results suggest that the variables' sensitivity is similar in all cases, i.e. varying the *number of items per bid*, the *number of providers* or the *number of bids per provider* leads to similar trends. Therefore, the stages that are more time-consuming are quite the same in every possible configuration: for instance, stage Δt10 is always the most time consuming, irrespective of which parameter is being varied. Moreover, we can observe similar trends for the rest of stages (from Δt1 to Δt10). Hence, it seems that the time distribution along the different stages can be regarded as being independent from the parameter setting.

Fig. 6 illustrates the average percentage, over all the performed trials, of the total time that each stage consumes. We observe that: (1) The Δ**t1**, Δ**t3**, Δ**t4**, Δ**t5**, Δ**t9**, Δ**t10** stages are the most time-consuming (92% of the total time). Since these stages involve ontology checking and message encoding and decoding, we can conclude that

[3] The way the times vary when increasing those parameters is not linear. Nonetheless, we did not deeply study this aspect, because the main issue for us was to assess the difference of these times with respect to the solver component time by itself.

these activities are a bottleneck. (2) The solver time (Δt7) is almost a negligible part of the total time. (3) Manipulating classes (stages Δt2, Δt6, Δt8 and Δt11) and solving the combinatorial problem (Δt7) is not as time-consuming as encoding and decoding messages and ontology objects.

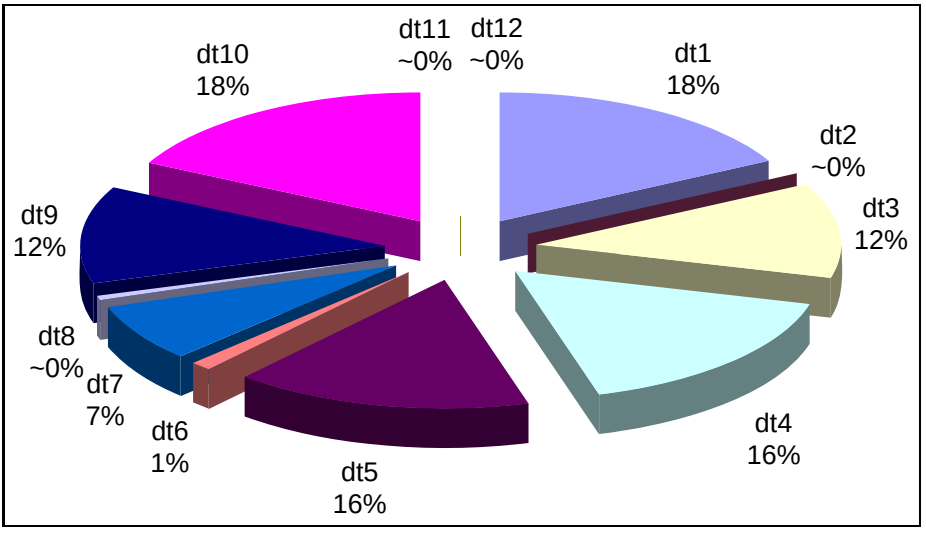

Fig. 6. Average times spent at the different evaluation stages

Fig. 8 depict the accumulated time spent on all stages for a collection of negotiation scenarios, which we refer to as the *total time*. More precisely, Fig. 8 depicts configurations whose total time lies between 30 and 50 seconds. It is conceivable to regard them as the edge values, although it is a very arbitrary matter. Some observations follow from analysing the figures above:

1. The agent-awareness of *iBundler* is costly. We observe that the percentage of total time employed to solve the winner determination problem is small with respect to agent related tasks.
2. Using the solver component we can easily solve problems of more than 2000 bids in less than one minute, whereas the agent service can handle in reasonable time less than 750 bids.
3. Therefore, small, and medium-size negotiation scenarios can be reliably tackled with *iBundler*. Nonetheless, time performance significantly impoverishes when handling large-size negotiation scenarios.

5.2 Memory Usage

In this case, we found similar results when comparing the *Solver* component with *iBundler*. The amount of memory required in the worst case is much the same for

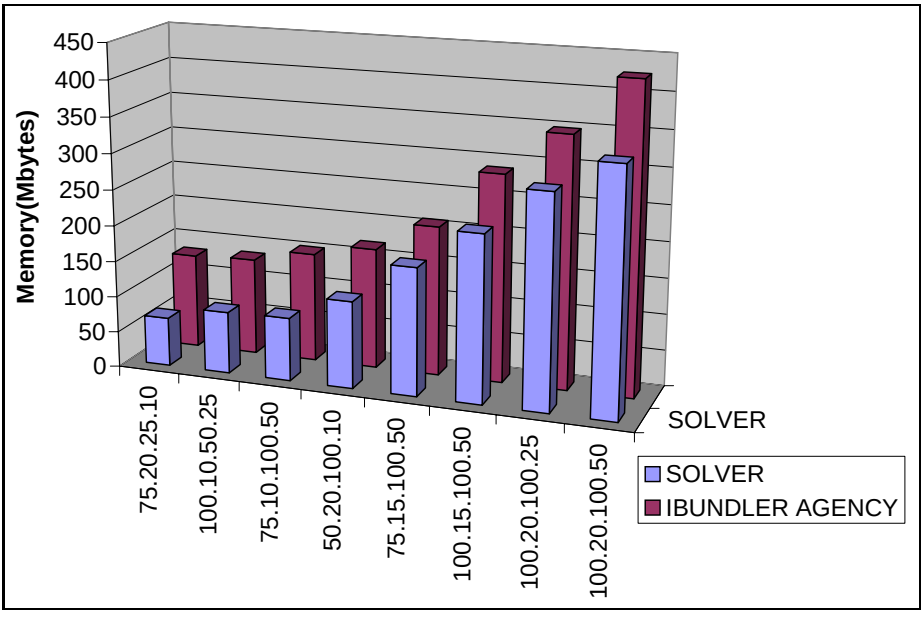

Fig. 7. Memory consumption

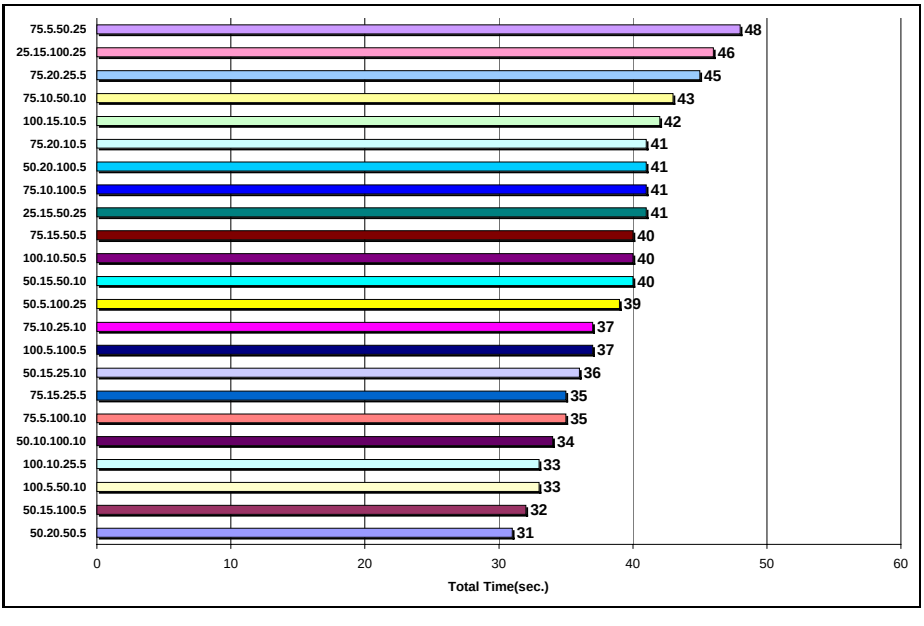

Fig. 8. Time performance for negotiation scenarios on the edge of acceptability

both cases. The memory consumption in both cases is highly dependent on the ontology structure. It is not surprising that the memory peak is similar in both cases, as the information quantity to be represented is actually the same. The largest amount of information is used to represent all the bids. Both *Solver* and JADE have to load in memory the information representing a problem, namely an RFQ and the received bids (the former as a Java object and the latter as a file containing matrices). Fig. 7 compares the memory use for the *iBundler* agency and *Solver*.

6 Conclusions

The tests we ran show that offering *iBundler* as an agent service implies a significant time overload, while the memory usage is only slightly affected. The main cause of such an overload is related to the encoding and the decoding of ontological objects and messages. The message serializations and deserializations, along with ontology checkings, heavily overload the system as the dimensions of the negotiation scenario grow. We propose several actions to alleviate this effect. Firstly, we have observed that the main amount of information is gathered in representing bids. Their presence in objects and messages is the foremost cause of *iBundler*'s time overload. Thus, a suitable work-around is to use, at ontology design time, a more synthetic bidding language, in which bids can be expressed more concisely; for instance, introducing a preprocessing phase in which equal (and even similar) bids are grouped, in order to obtain a more compact representation. The resulting ontology would generate more tractable objects. Secondly, it would be also helpful to improve the performances of the JADE modules devoted to the ontology checking and serialization processes. All in all, *iBundler* can satisfactorily handle small and medium-size negotiation scenarios. Thus, although the automation of the negotiation process with agents helps in saving time in managing negotiations, the scalability in terms of time response of *iBundler* is limited.

As future work, we propose a comparison of *iBundler* with other distributed solutions such as CORBA (http://www.corba.org) or JAVA RMI (http://java.sun.com). Nonetheless, we should notice that agent technology offers a higher level of abstraction, and thus we would lose the transparency and portability offered by the agent paradigm.

We conclude that, while agent technology adds a higher level of abstraction and eases inter-platform communication, state-of-the-art agent technologies require further improvements to tackle real-world domains.

Acknowledgments

This work has been funded by the Spanish Science and Education Ministry as part of the Web-i2 (TIC-2003-08763-C02-00) and IEA (TIN2006-15662-C02-01) projects, and by the Spanish Council for Scientific Research as part of the 20065 OI 099 project. Andrea Giovannucci enjoys the BEC.09.01.04/05164 CSIC scholarship.

References

1. Giovanucci, A., Rodríguez-Aguilar, J.A., Reyes-Moro, A., Noria, F.X., Cerquides, J.: Towards automated procurement via agent-aware negotiation support. In: Third International Joint Conference on Autonomous Agents and Multiagent Systems, New York (2004) 244–251
2. Reyes-Moro, A., Rodríguez-Aguilar, J.A., López-Sánchez, M., Cerquides, J., Gutiérrez-Magallanes, D.: Embedding decision support in e-sourcing tools: Quotes, a case study. Group Decision and Negotiation **12** (2003) 347–355
3. Brazier, F., van Steen, M., Wijngaards, N.: On MAS scalability. In: Proceedings of Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS, Montreal (2001) 121–126
4. Deters, R.: Scalability & multi-agent systems. In: Proceedings of Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS, Montreal (2001)
5. Kahn, M.L., Della Torre Cicalese, C.: COABS grid scalability experiments. Autonomous Agents and Multi-Agent Systems **7** (2003) 171–178
6. Klein, M., Rodriguez-Aguilar, J.A., Dellarocas, C.: Using domain-independent exception handling services to enable robust open multi-agent systems: The case of agent death. Autonomous Agents and Multi-Agent Systems **7** (2003) 179–189
7. Fedoruk, A., Deters, R.: Improving fault-tolerance by replicating agents. In: AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems, ACM Press (2002) 737–744
8. Yoo, M.J.: An industrial application of agents for dynamic planning and scheduling. In: AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems, ACM Press (2002) 264–271
9. Jarrar, M., Meersman, R.: Scalability and knowledge reusability in ontology modeling. In: Proceedings of the International conference on Infrastructure for e-Business, e-Education, e-Science, and e-Medicine. Volume SSGRR2002s., Rome, SSGRR education center (2002)
10. Wache, H., Serafini, L., García-Castro, R.: D2.1.1 survey of scalability techniques for reasoning with ontologies. Technical report, Knowledge Web (2004)
11. OMG, FIPA: OMG and FIPA standardisation for agent technology: competition or convergence? http://www.cordis.lu/infowin/acts/analysys/products/thematic/agents/ch2/ch2.htm (1999)
12. Sandholm, T., Suri, S., Gilpin, A., Levine, D.: Winner determination in combinatorial auction generalizations. In: First Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02), Bologna (2002) 69–76
13. Hillier, F.S., Liberman, G.J. In: Introduction to Operations Research. Mc Graw Hill (2001) 576–653
14. Odell, J., van Dyke Parunak, H., Bauer, B.: Extending UML for agents. In: Proceedings of the Agent-Oriented Information Systems Workshop, Austin, TX, 17th National Conference on Artificial Intelligence (2000) 3–17
15. FIPA: FIPA interaction protocol library specification. Technical Report DC00025F, Foundation for Intelligent Physical Agents (2003)

OWL-P: A Methodology for Business Process Development

Nirmit Desai[1], Ashok U. Mallya[2], Amit K. Chopra[1], and Munindar P. Singh[1]

[1] Department of Computer Science
North Carolina State University, Raleigh, NC 27695-8206, USA
{nvdesai, aumallya, akchopra, singh}@ncsu.edu
[2] Veraz Networks Inc
926 Rock Avenue, Suite 20, San Jose, CA 95131, USA
amallya@veraznet.com

Abstract. Business process modelling and enactment are notoriously complex, especially in open settings where the business partners are autonomous, requirements must be continually finessed, and exceptions frequently arise because of real-world or organizational problems. Traditional approaches, which attempt to capture processes as monolithic flows, have proved inadequate in addressing these challenges. We propose an agent-based approach for business process modelling and enactment which is centred around the concepts of commitment-based agent interaction protocols and policies. A (business) protocol is a modular, public specification of an interaction among different roles. Such protocols, when integrated with the internal business policies of the participants, yield concrete business processes. We show how this reusable, refinable and evolvable abstraction simplifies business process design and development.

1 Introduction

Unlike traditional business processes, processes in open, Web-based settings typically involve complex interactions among autonomous, heterogeneous *business partners*. Conventionally, business processes are modelled as monolithic workflows, specifying exact steps for each participant. Because of the exceptions and opportunities that arise in open environments, business relationships cannot be pre-configured to the full detail. Thus, flow-based models are difficult to develop and maintain in the face of evolving requirements. Furthermore, such conventional models do not facilitate flexible actions by the participants.

This paper proposes an approach for business process modelling and enactment, which is based on a combination of protocols and policies. The key idea is to capture meaningful interactions as *protocols*. Protocols can involve multiple roles and address specific purposes such as ordering, payment, shipping and so on. Protocols are given a contractual semantics in terms of commitments among roles that capture the essence of the relationship among roles. In order to maximize participants' autonomy and to be reusable, protocols emphasize the essence of the interactions and omit local details. Such details are supplied by each participant's *policies*. For example, when a protocol allows a participant to choose from multiple actions, the participant's policy decides

M. Kolp et al. (Eds.): AOIS 2005, LNAI 3529, pp. 79–94, 2006.

which one to perform. Typically, policies are business logic that provide and process message contents.

This paper seeks to develop the main techniques needed to make this promising approach practical. Our contributions include a language and an ontology for protocols called OWL-P, which is coded in the Web Ontology Language (OWL) [1]. OWL-P describes concepts such as roles, the messages exchanged between the roles, and declarative protocol rules. OWL-P compiles into Jess rules which then can be integrated with the local policies in a principled manner.

Protocols are not only reusable across business processes but also amenable to abstractions such as refinement and aggregation [2]. The key benefits of this approach are (1) a separation of concerns between protocols and policies in contrast to traditional monolithic approaches; and (2) reusability of protocol specifications based on design abstractions such as specialization and aggregation.

1.1 Running Example

As a running example, let's consider a business process involving a small number of parties. Fig. 1 depicts a purchase process where items to be purchased have already been selected and the price has been agreed upon. Each participant is shown by a separate shaded region, the graph made of dark edges denotes the *flow* of the given participant. Circular nodes represent the participant's internal business logic or policies, e.g. to decide the parameters of an out-bound message. Rectangular nodes represent external interfaces through which a participant receives messages. Thus, an ordering of dark arrows, circles and rectangles represents the local process of the participant. When there

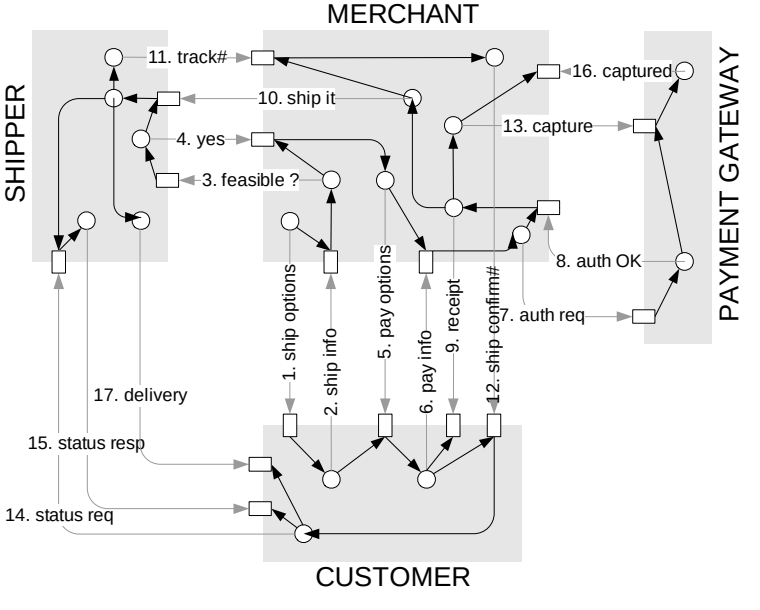

Fig. 1. A purchasing process

are multiple out-edges from a node, all of them are taken concurrently. The messages are labelled with numbers to indicate a possible order in which they might occur.

1.2 Shortcomings of Traditional Approaches

The process of Fig. 1 can be captured via a traditional flow-based modelling approach. Such a representation would be functionally correct, but inadequate from the perspectives of open environments. The following are its shortcomings:

Lack of Contractual Semantics. Traditional approaches expose low-level interfaces, e.g. via WSDL [3], but associate no contractual semantics with the participants' actions. To control the autonomy of the participants and enforcing compliance, such a semantics is crucial. This lack precludes flexible enactment (as needed to handle exceptions) as well as reliable compliance checking. For this reason, we cannot determine if a deviation from a specific sequence of steps is significant.

Lack of Reusable Components. The local processes of the partners are not reusable even though the patterns of interaction among the participants might be. Local processes are monolithic in nature and formed by *ad hoc* intertwining of internal business logic and external interactions. Since business logic is proprietary, local processes of one partner are not usable by another. For instance, if a new customer were to participate in this open environment, its local process would have to be developed from scratch.

Organization

Section 2 introduces some key concepts and terminology. Section 3 describes our protocol specification language and its semantics. Section 4 discusses composite protocols and their construction. Section 5 shows how augmenting policies with protocols can be used to develop processes. Section 6 compares our work with relevant research efforts in the area and Section 7 concludes the paper.

2 Concepts and Terminology

Fig. 2 shows our conceptual model for treatment of business processes based on protocols and policies. Boxed rectangles are abstract entities (interfaces), which must be combined with business policies to yield concrete entities that can be fielded in a running system (rounded rectangles). Abstract entities should be published, shared and reused among the process developers. We specify a business protocol using rules termed *protocol logic* that specify the interactions of the participating *roles*. Roles are abstract and are adopted by agents to enable concrete computations. Whereas the protocol logic specifies the protocol from the global perspective, a *role skeleton* specifies the protocol from the perspective of the corresponding participant role. Thus, each role skeleton defines the behaviour of the respective role in a protocol.

When an agent needs to participate in multiple protocols, a *composite skeleton* can be constructed by combining the protocols according to some composition constraints and deriving the role skeleton. For example, in a supply chain process, a supplier would be

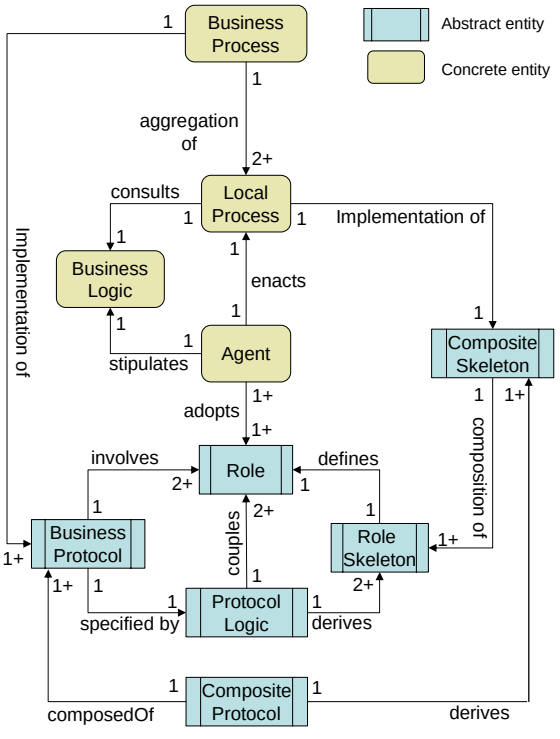

Fig. 2. Conceptual model

a merchant when interacting with a retailer in a trading protocol and would be an item-sender in a shipping protocol for sending goods to the retailer. A composite skeleton for such a supplier could be composed by combining trading and shipping protocols and then deriving the role skeleton for item-sender/merchant role. The resultant composite skeleton could also be published and then reused for developing local processes of other suppliers.

An agent's private policies or *business logic* are described via rules. The *local process* of an agent is an executable realization of a composite skeleton obtained by integration of the protocol logic of the composite skeleton and the business logic of the agent. A *business process* is the aggregation of the local processes of all the agents participating in it. Conversely, a business process is an implementation of the constituent business protocols.

2.1 Protocols and Commitments

Commitments are used to give semantics to agent interaction. As agents interact, they create and manipulate commitments. A commitment $\mathsf{C}(x, y, p)$ denotes that agent x is obligated to agent y for bringing about condition p. Commitments can be *conditional*, denoted by $\mathsf{CC}(x, y, p, q)$, meaning that x is committed to y to bring about q if p holds

where p is called the precondition of the commitment. For example, the conditional commitment $\mathsf{CC}(c, b, \mathit{goods}(g), \mathit{pay}(p))$ means that the customer c is committed to pay the bookstore b an amount p if the bookstore delivers the book g to the customer. Commitments are created, satisfied and transformed in certain ways [4]. The following are the operations defined on commitments:

Op1. $\textsc{create}(x, c)$ establishes the commitment *c* in the system.
Op2. $\textsc{cancel}(x, c)$ cancels the commitment *c*.
Op3. $\textsc{release}(y, c)$ releases *c*'s debtor from commitment c without c being fulfilled.
Op4. $\textsc{assign}(y, z, c)$ replaces *y* with *z* as *c*'s creditor.
Op5. $\textsc{delegate}(x, z, c)$ replaces *x* with *z* as the *c*'s debtor.
Op6. $\textsc{discharge}(x, c)$ c's debtor x fulfils the commitment.

A commitment is said to be *active* if it has been created but not yet discharged. The rules regarding discharge of a commitment are given below.

Dis1. $\frac{C(x,y,p) \wedge p}{discharge(x, C(x,y,p))}$
Dis2. $\frac{CC(x,y,p,q) \wedge p}{create(x, C(x,y,q)) \wedge discharge(x, CC(x,y,p,q))}$
Dis3. $\frac{CC(x,y,p,q) \wedge q}{discharge(x, CC(x,y,p,q))}$

3 Protocol Specification

A business protocol is a specification of the allowed interactions between two or more participant roles. The specification focusses on the interactions and their semantics. What does it mean to send a certain message to a business partner? What is expected of the participants wishing to comply to a business protocol? How are the protocols specified? These are the questions we address in this section.

3.1 OWL-P: OWL for Protocols

OWL-P is an ontology based on OWL for specifying protocols; it functions as a schema or language for protocols. The main computational aspects of protocols are specified using rules. We employ the Semantic Web Rule Language (SWRL) [5] for defining rules. SWRL allows us to specify implication rules over entities defined as OWL-P instances. The availability of tools such as Protégé [6] is a motivation for grounding our approach in OWL.

The important OWL-P elements and their properties are shown in Fig. 3. An entity with a little rectangle represents the domain of the corresponding property. Many of the properties are self-explanatory and reflect the conceptual model introduced in Section 2.

Slots are analogous to data variables. A slot is said to be *defined* when it is assigned a value and it said to be *used* when its value is assigned to another slot. A slot in a protocol may be assigned a value produced by another protocol and hence be represented as an *External Slot*. An external slot is untyped until it is given the type of the external value to which it is bound. By contrast, a *Native Slot* is typed and defined inside the protocol.

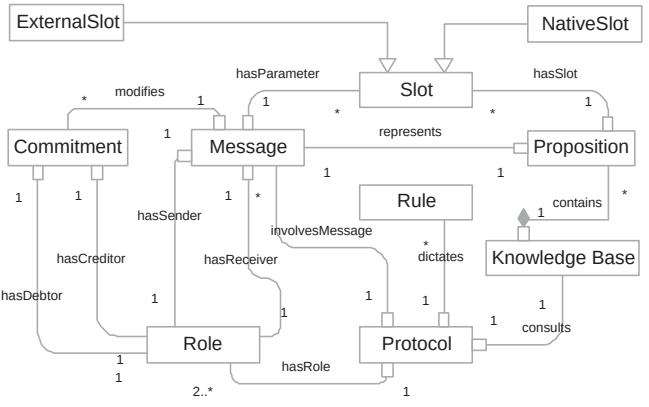

Fig. 3. Basic OWL-P ontology

A *Protocol* dictates several rules and consults a *Knowledge Base*. A knowledge base consists of a set of *Propositions*. A proposition in a knowledge base may correspond to a message, active commitments or other domain specific propositions.

Fig. 4 shows a protocol for ordering goods (along with others, to which we refer later). For readability, a leading and trailing * is placed around external slot names, as in *amount* and *itemID*. The customer requests a quote for an item, to which the merchant responds by providing a quote. Here, a commitment is created providing semantics for the message. The commitment means that the merchant guarantees receipt of the item if the customer pays the quoted price. The customer can either accept the quote or reject it (not shown). Again, the semantics of acceptance is given by the creation of another commitment from the customer to pay the quoted price if it receives the requested item. Below are the rules for the Order protocol in the "antecedents $\Rightarrow$ consequents" notation.

Ord1. contains(KB, startProp) $\Rightarrow$ send(B, reqForQuote(?itemID))
Ord2. contains(KB, reqForQuoteProp(?itemID)) $\Rightarrow$ send(S, quote(?itemID, ?itemPrice)) $\wedge$ createCommitment(S, CC(S, B, pay(?itemPrice), goods(?itemID)))
Ord3. contains(KB, quoteProp(?itemID, ?itemPrice))$\Rightarrow$send(B, acceptQuote(?itemID, ?itemPrice)) $\wedge$ createCommitment(C, CC(C, M, goods(?itemID), pay(?itemPrice)))
Ord4. contains(KB, quoteProp(?itemID, ?itemPrice))$\Rightarrow$send(B, acceptQuote(?itemID, ?itemPrice))

In the above rules, reqForQuote, quote, and acceptQuote are OWL-P message instances (individuals in OWL terminology). Corresponding proposition instances are reqForQuoteProp, quoteProp, and acceptQuoteProp. Propositions pay and goods are commitment conditions, while itemID and itemPrice are native slots. Readers may notice that the itemID variable in the first rule is not assigned any value by the antecedents. It means that the rule is abstract and not executable and, as we will see in Section 5.2, it can be augmented with business logic that produces such values. Rules having unde-

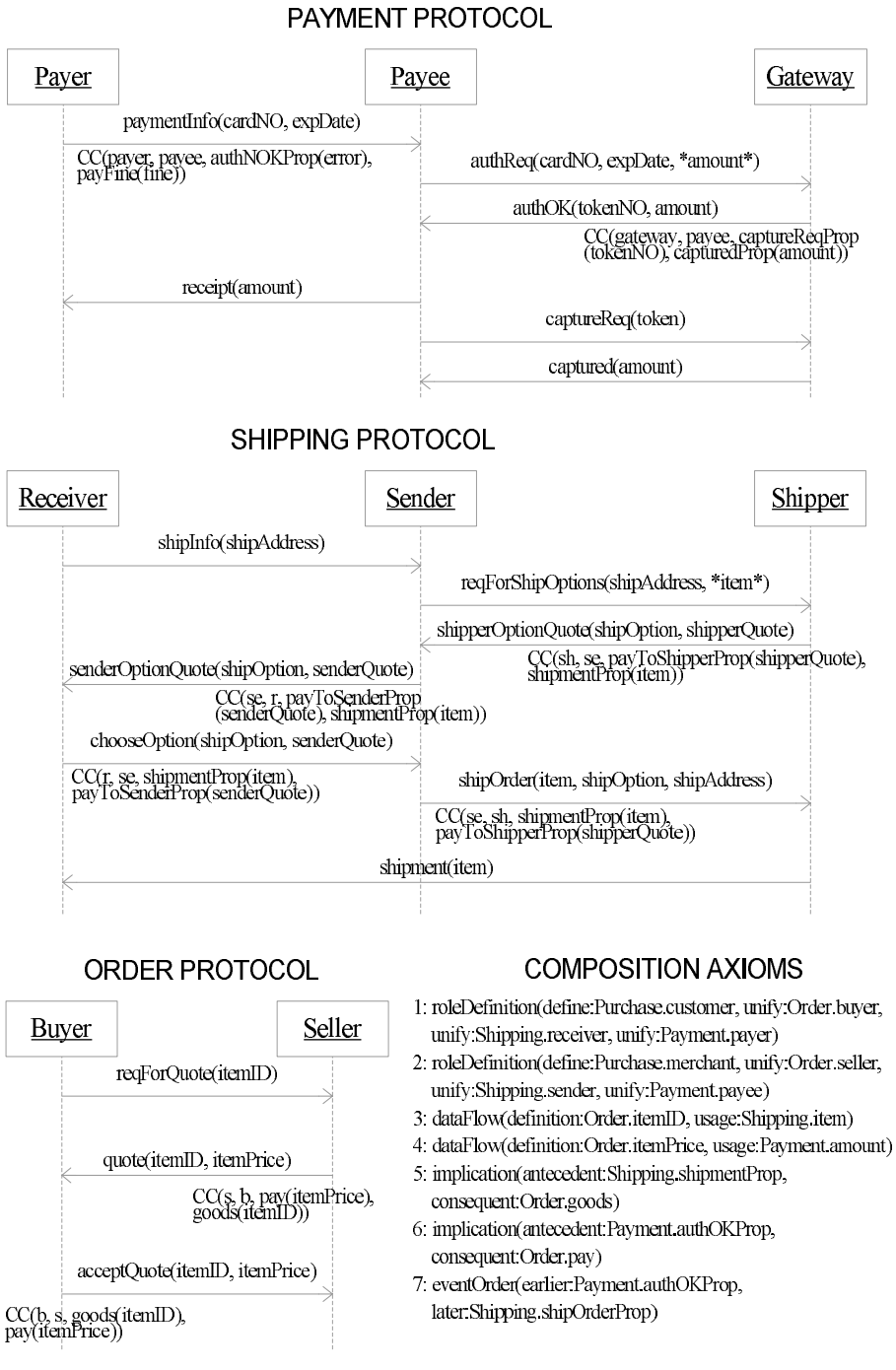

Fig. 4. Example: Order, Shipping, and Payment protocols and their composition

fined native slots must be augmented with the business logic that produces such values. How do these rules define the protocol? The next section describes the operational semantics of the protocol rules. The OWL-P ontology and protocol instance examples in their RDF/XML serialization, and corresponding Protégé projects are available on the Web [7].

3.2 Operational Semantics

Protocols are specified from the global perspective with an assumption of an abstract global knowledge base and the rules are assumed to be forward-chained. OWL-P defines several property predicates with operational semantics. Table 1 lists the seman-

Table 1. Operational semantics of protocol rules

Predicate	Domain	Range	Meaning
contains	KB	Proposition	Proposition $\in$ KB ?
assert	Proposition	KB	KB $\leftarrow$ KB $\cup$ Proposition
send	Role	Message	Asynchronous send to the receiver assert(KB, MessageProp)
receive	Role	Message	Asynchronous receive from the sender assert(KB, MessageProp)
createComm	Role	Commitment	assert(KB, CommitmentProp)

tics for such property predicates of OWL-P. A proposition cannot be retracted from a knowledge base. In the forthcoming examples, we may omit the OWL-P properties, e.g. contains, send, createCommitment when the meaning is clear. Fig. 5 shows an inside view of an agent to demonstrate how the rules govern the interactions. For now, ignore steps 3, 4, and 5 dealing with policy rules. When a message is received, it is checked against the protocol rules to see if it may be consumed. If so, a corresponding proposition is asserted and any activated rules are executed. Doing so may activate other rules, resulting in further propositions being asserted and messages being sent.

4 Composite Protocols

The previous section described how to specify individual protocols. To meet the requirements of business processes, it is necessary to compose them from simpler protocols. Now we show how protocols can be composed.

Conceptually, each component protocol achieves a business goal. Thus, several such protocols composed together would achieve the goals of the larger business process. Composition also enables refinements of protocols with additional rules. The ability to compose protocols would allow significant reuse of published protocols. How can we construct such composite protocols? How do they facilitate reusability? How do they allow refinements of protocols? This section answers these questions.

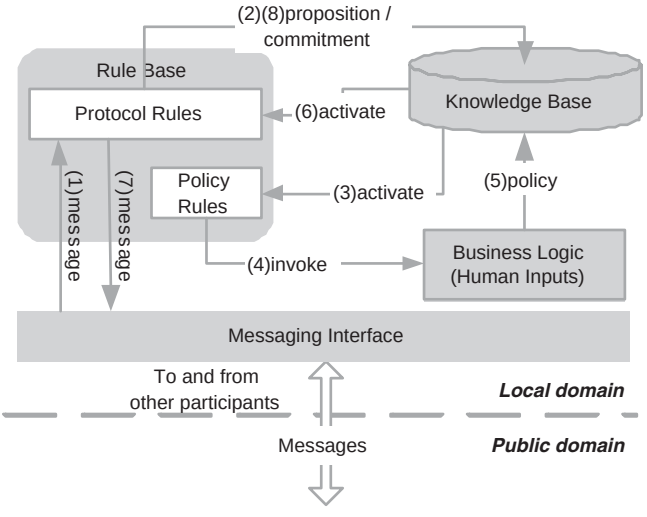

Fig. 5. Agent architecture: protocol and policy interplay

4.1 Construction of Composite Protocols

Fig. 6 describes the OWL-P classes and properties that deal with the problem of protocol composition. A *Composite Protocol* is an aggregation of component protocols and is defined by a *Composition Profile*. A composition profile describes the combination of two or more protocols by stipulating several *Composition Axioms*. Composition axioms

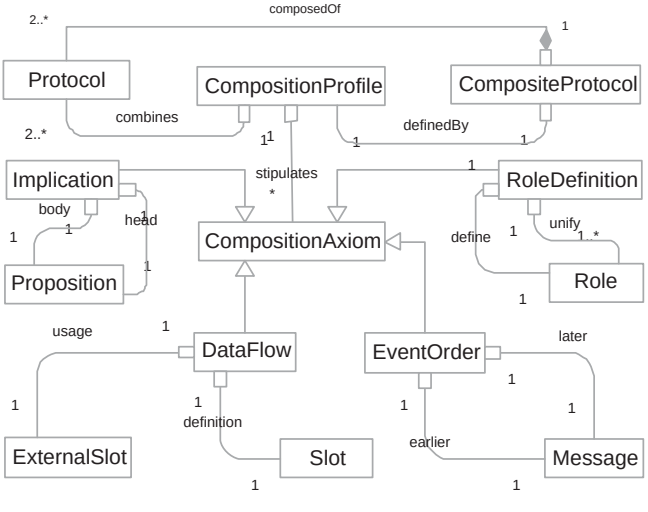

Fig. 6. OWL-P composition classes and properties

define relationships among the protocols being combined. The operational semantics of an axiom specifies the way in which the relationships affect the composite protocol. Fig. 4 depicts an Order protocol, a Shipping protocol, a Payment protocol and a set of composition axiom instances stating the relationships among them.

A *Role Definition* axiom states which of the roles in the component protocols are to be adopted by the same agent and defines the name of the unified (coalesced) role in the composite protocol. In the example, the first axiom states that the roles of a customer in Order, a payer in Payment and a receiver in Shipping protocol are played by an agent who will play the role of a customer in the Purchase protocol.

A *Data Flow* axiom states a data-flow dependency among the protocols. A component protocol might be using a slot defined by another component protocol, possibly with a different name. Since a slot can be defined only once, and native slots must be defined inside the protocol, they cannot use a value defined by another protocol. Hence, the range of the *usage* property must be an external slot. In the example, the fourth axiom states that the slot amount in the Payment protocol gets its value from the slot itemPrice in the Order protocol. Such a dependency exerts an ordering among the rule defining the slot and all the rules using it: none of the the rules using the slot can fire before the slot is assigned a value by the defining rule.

An *Implication* axiom states that an assertion of proposition X in a component protocol implies an assertion of proposition Y in another component protocol. For example, the sixth axiom states that an assertion of authOKProp in the Payment protocol means an assertion of pay in the Order protocol. This can be easily achieved by adding an implication rule to the composite rulebase.

Unlike the DataFlow axiom, an *EventOrder* axiom explicitly specifies an ordering among the messages of the component protocols. For example, the seventh axiom states that an authOK message from the payment gateway must be received before a shipOrder message is sent to the shipper. This can be achieved by making the rule for the later event depend on the rule for the earlier event.

Operational semantics of these axioms are given in [8]. Composition axioms have to be specified by a designer. There might be several ways of composing the component protocols yielding different composite protocols. As a special case, if the component protocols are completely independent of each other, no axioms need be specified and their OWL-P specifications can be simply aggregated yielding the OWL-P specification of the composite protocol. If deemed necessary, more types of composition axiom can be defined along with their properties and operational semantics. A composite protocol exposes its compositionProfile and possesses all the properties of the component protocols. Hence, a composite protocol itself can be a component protocol in some other composition profile instance. How can we determine whether additional component protocols are needed? To answer this question, we define *closed* and *open* protocols. A protocol is closed if it has no external slots, and all the commitments created in the protocol can be discharged by the protocol. A protocol is *open* if it is not closed. A designer's goal is to obtain a closed protocol by repeated applications of composition. Observe that, in Fig. 4, the Order protocol is open as its rules do not assert propositions pay and goods necessary for discharging the commitments created. The Payment,

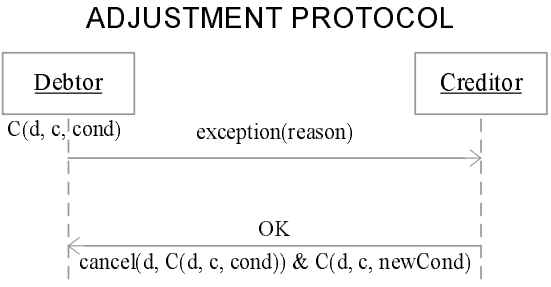

Fig. 7. Handling refinements by composition

Shipping and Purchase protocols are also open according to the definition. A designer would choose protocols that assert these missing propositions and combine them with the Purchase protocol to obtain a closed composite protocol.

4.2 Refinement by Composition

Business protocols evolve continually as new requirements and new features routinely arise. Therefore, the ability to systematically refine protocols is valuable. In the composite Purchase protocol, consider a situation in which the customer has already paid the merchant for the goods and hence the commitment C(S,B,goods(itemID)) is active. However, while trying to order the shipment, if a fire destroys the merchant's warehouse, the merchant will not be able to honor its commitment to ship the item. How can such exceptions be handled? The protocol could detect the violation due to an unfulfilled commitment and the merchant could be held legally responsible. However, A more flexible solution would be to allow the merchant to refund money and cancel its commitment to ship, provided the customer agrees to it. We can achieve this flexibility by combining the purchase protocol with the adjustment protocol shown in Fig. 7 yielding the composite protocol *Flexible* with these composition axioms:

AdjAx1. roleDefinition(define: Flexible.customer, unify: Purchase.customer, unify: Adjustment.creditor)
AdjAx2. roleDefinition(define: Flexible.merchant, unify: Purchase.merchant, unify: Adjustment.debtor)
AdjAx3. implication(body: Purchase.C(B,S,goods(itemID)), head: Cancel.C(D,C, cond))
AdjAx4. implication(body: Cancel.C(D,C,newCond), head: Purchase.C(S,B,refund))

Similar protocols for assigning, delegating, and releasing commitments can be defined. Adding new functionalities would involve composition of a set of rules for the new requirements with the original protocol.

5 Processes

As described in Section 2, a process is an aggregation of the local processes of participating agents. However, an OWL-P specification of a protocol is a model of the interaction from a global perspective. To construct the local process of a participant, we need to derive the participant's view of the protocol, called its *role skeleton*. Section 5.1 describes the generation of role skeletons from an OWL-P specification.

5.1 Role Skeletons

A role skeleton is one role's view of the protocol. Here, we provide the intuition behind generating role skeletons from an OWL-P protocol specification. The complete algorithm is given in [8]. OWL-P describes a protocol from the global perspective where the propositions are added to the global state and there are no distributed sites. As in all distributed systems, the state of a protocol as seen by a role is changed only when a message is sent or received by that role. This observation forms the basis for deriving role skeletons.

As an example, we show a rule in the Shipping protocol in Fig. 4, and the same rule in the generated skeleton of the receiver. As the receiver would not be aware of the previous exchanges between the sender and the shipper, the antecedent of the rule for receiving senderOptionQuote should be adjusted as shown below.

Protocol Rule
shipperOptionQuoteProp(.,.) ⇒ senderOptionQuote(.,.) ∧
CC(Se,Re,payToSenderProp(.),shipmentProp(.))

Receiver Skeleton Rule
shipInfoProp(?shipAddress) ⇒ receive(senderOptionQuote(.,.))
∧ CC(Se,Re,payToSenderProp(.),shipmentProp(.))

5.2 Policies

Generation of a role skeleton is not enough to obtain a local process of a participant. As we mentioned earlier, some of the rules of the protocols may be abstract, meaning that values of some of the native slots in the rule must be produced by the role's business logic. Hence, a role skeleton must be augmented with the business logic to obtain a local process. How can we determine whether an augmented role skeleton is a local process? To answer this question, we first define *concrete* and *abstract* role skeletons, as well as a *local process*. A role skeleton is *concrete* if all of its native slots are defined. A role skeleton is *abstract* if it is not concrete. A *local process* is a role skeleton that is concrete and derived from a closed protocol.

Seller skeleton rules:
startProp ⇒ receive(C, reqForQuote(?itemID))

reqForQuoteProp(?itemID) ∧ **quotePolicy(?itemPrice)** ⇒
quote(?itemID, ?itemPrice) ∧ CC(S, B, pay(?itemPrice), goods(?itemID))

quoteProp(?itemID, ?itemPrice) ⇒ receive(C, acceptQuote(?itemID, ?itemPrice)) ∧
CC(C, M, goods(?itemID), pay(?itemPrice))

Seller policy rule for quote:
reqForQuoteProp(?itemID) ⇒ **call(policyDecider, quotePolicy(?itemID))**

We propose that the business logic be specified in terms of the local policy rules of the agents. The skeleton of the merchant role in the Order protocol augmented with the policy rules of the seller agent is shown above. The last rule is the policy rule that calls a business logic operation to decide how much to quote. The operation would assert the quotePolicy proposition and that would activate the second protocol rule. Observe that this pattern of augmenting policy rules is general and will be applied to the rules where the agent has to make a decision and respond. It would also assign a value to native slots that are not defined.

5.3 Usage

Fig. 8 summarizes our methodology with a scenario involving a customer interested in purchasing goods online. Software designers design protocols and register them with protocol repositories. They may also construct composite protocols and reuse the existing component protocols from the repository. A merchant wishing to sell goods online looks up the repository for a suitable Purchase protocol. It generates the skeleton for the merchant role, augments it with its local policies and deploys the result as a service. The service profile for this service would contain an OWL-P description of the Purchase protocol. The service can be registered with a UDDI registry. If a customer wishes to buy goods online, it searches the UDDI registry, finds the merchant and acquires the OWL-P skeleton for the customer role from the merchant. The customer enacts its local process by augmenting the skeletons with its local policies and starts interacting with the merchant. We have developed tools to support these development scenarios and a prototype implementation based on the agent architecture of Fig. 5 [9]. Note that we propose only a methodology for development and there might be other issues to be resolved for realizing an e-commerce enterprise.

6 Related Work

Several areas of research are relevant to our work. We discuss each of them briefly and highlight the differences.

Composition. BPEL [10] is a language designed to specify the static composition of Web services. However, it mixes the interaction activities with the business logic making it unsuitable for reuse. OWL-S [11], which includes a process model for Web Services, uses semantic annotations to facilitate dynamic composition. A composed service is produced at runtime based on constraints. While dynamic service composition has some advantages, it assumes a perfect markup of the services being composed. Dynamic composition in OWL-S involves ontological matching between inputs and outputs. Such a matching might be difficult to obtain automatically given the heterogeneity of the web. For this reason, we do not emphasize dynamic service composition.

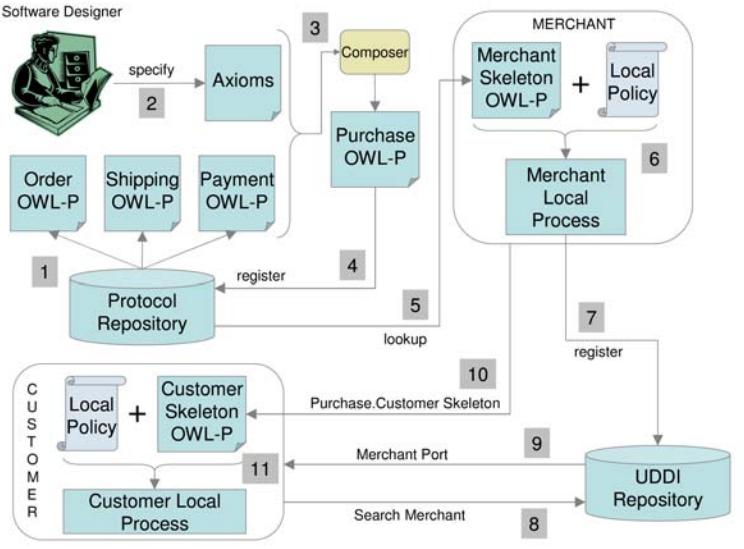

Fig. 8. Usage scenario

Our goal is to provide a human designer with tools to facilitate service composition. Unlike BPEL, which specifies the internal orchestration of services, WSCI [12] specifies the conversational behaviour of a service using control flow constructs. However, these specifications lack a semantics, which makes them difficult to compose and reuse.

Several other approaches aim to solve the service composition problem by emphasizing formal specifications to achieve verifiability. Solanki *et al.* [13] employ interval temporal logic to specify and verify ongoing behaviour of a composed service. Their use of "assumption" and "commitment" (different meaning than here) assertions allows better compositionality. Gerede *et al.* [14] treat services as activity-based finite automata to study the decidability of composability and existence of a look-ahead delegator given a set of existing services. However, these approaches consider neither the autonomy of the partners, nor the flexibility of composition.

Software Engineering. Our methodology advocates and enables reuse of protocols as building blocks of business processes. Protocols can not only be composed, they can also be systematically refined to yield more robust protocols. Mallya and Singh [2] treat these concepts formally. The MIT Process Handbook [15], in a similar vein, catalogues different kinds of business processes in a hierarchy. For example, *sell* is a generic business process. It can be qualified by *sell what*, *sell to who*, and so on. Our notion of a protocol hierarchy bears some similarity to the Handbook. RosettaNet [16] is similar to our approach in that it centres around publishing protocols and designing the business processes around them. However, it is currently limited to two-party request-response interactions called Partner Interface Processes (PIPs) and, more importantly, PIPs lacks a formal semantics.

Agent-oriented software methodologies aim to apply software engineering principles in the agent context e.g. Gaia, KAOS, MaSE, and SADDE [17]. Tropos [18] differs from these in that it includes an early requirements stage in the process. Gaia [19] differs from others in that it describes roles in the software system being developed and identifies processes in which they are involved as well as safety and liveness conditions for the processes. It incorporates protocols under the *interaction model* and can be used with commitment protocols. Baïna *et al.* [20] advocate a model-driven Web service development approach to ensure compliance between a service's implementation and its external protocol specifications. Our work differs from these in that it is aimed at achieving protocol re-usability by separation of protocols and policies and it addresses the problem of protocol compositions.

7 Conclusions

We have presented an approach for designing processes that recognizes the fundamental interactive nature of open environments where the autonomy of the participants must be preserved. Commitments provide the basis for a semantics of the actions of the participants, thereby enabling the detection of violations. The significance of this work derives from the importance of processes in modern business practice. With over 100 limited business protocols having been defined [16], this approach will permit the development and usage of an ever-increasing set of protocols for critical business functions. We demonstrated the practicality of our approach by embedding it in an ontology and language for specifying protocols. Not only is this approach conducive to reuse, refinement and aggregation but it has also been implemented in a prototype tool. It would be ineteresting to see theoretical foundations of this work in the process algebra. It would allow one to establish properties of the protocols and relationships among them.

Acknowledgments

This research was sponsored by NSF grant DST-0139037 and a DARPA project.

References

1. OWL Web Ontology Language: Overview. www.w3.org/TR/owl-features/ (2004) W3C Recommendation.
2. Mallya, A.U., Singh, M.P.: An algebra for commitment protocols. Autonomous Agents and Multiagent Systems (2006) http://dx.doi.org/10.1007/s10458-006-7232-1.
3. WSDL: Web Services Description Language (2002) http://www.w3.org/TR/wsdl.
4. Singh, M.P.: An ontology for commitments in multiagent systems: Toward a unification of normative concepts. Artificial Intelligence and Law **7** (1999) 97–113
5. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A semantic web rule language combining OWL and RuleML (May, 2004 (W3C Submission)) http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/.
6. Protégé: The Protégé ontology editor and knowledge acquisition system (2004) http://protege.stanford.edu/.

7. OWL-P Examples: (Business protocols modeled with owl-p) http://research.csc.ncsu.edu/mas/OWL-P/.
8. Desai, N., Mallya, A.U., Chopra, A.K., Singh, M.P.: Interaction protocols as design abstractions for business processes. IEEE Transactions on Software Engineering **31** (2005) 1015–1027
9. OWL-P Project: (Software, tools, and documentation) http://projects.semwebcentral.org/projects/owlp/.
10. BPEL: Business process execution language for web services, version 1.1 (2005) www-106.ibm.com/developerworks/webservices/library/ws-bpel.
11. DAML Services Coalition: DAML-S: Web service description for the semantic Web. In: Proceedings of the 1st International Semantic Web Conference (ISWC). (2002)
12. WSCI: Web service choreography interface 1.0 (2002) wwws.sun.com/ software/ xml/ developers/ wsci/ wsci-spec-10.pdf.
13. Solanki, M., Cau, A., Zedan, H.: Augmenting semantic web service descriptions with compositional specification. In: Proceedings of the International World Wide Web Conference. (2004) 544–552
14. Gerede, C.E., Hull, R., Ibarra, O., Su, J.: Automated composition of e-services: Lookaheads. In: Proceedings of the International Conference on Service Oriented Computing. (2004)
15. Malone, T.W., Crowston, K., Herman, G.A., eds.: Organizing Business Knowledge: The MIT Process Handbook. MIT Press, Cambridge, MA (2003)
16. RosettaNet: Home page (1998) www.rosettanet.org.
17. Bergenti, F., Gleizes, M.P., Zambonelli, F., eds.: Methodologies and Software Engineering for Agent Systems. Kluwer (2004)
18. Bresciani, P., Perini, A., Giorgini, P., Guinchiglia, F., Mylopolous, J.: Tropos: An agent-oriented software development methodology. Autonomous Agents and Multi-Agent Systems **8** (2004) 203–236
19. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Developing multiagent systems: The Gaia methodology. ACM Transactions on Software Engineering Methodology **12** (2003) 317–370
20. Baïna, K., Benatallah, B., Casati, F., Toumani, F.: Model-driven web service development. In: Proceedings of Advanced Information Systems Engineering: 16th International Conference, CAiSE. (June 2004)

Identification of Reusable Method Fragments from the PASSI Agent-Oriented Methodology

B. Henderson-Sellers[1], J. Debenham[1], Q.-N.N. Tran[1], M. Cossentino[2], and G. Low[3]

[1] University of Technology, Sydney, P.O. Box 123, Broadway, NSW 2007, Australia
{brian, debenham}@it.uts.edu.au, numitran@yahoo.com
[2] ICAR - Consiglio Nazionale Ricerche, Italy
cossentino@pa.icar.cnr.it
[3] University of New South Wales, NSW 2052, Australia
g.low@unsw.edu.au

Abstract. Theoretical proposals for the development of reusable method fragments are applied to the identification of method fragments in the agent-oriented methodology, PASSI. The format of these fragments is ensured as compatible with the structure and format already established for the OPEN Process Framework's (OPF) repository, which uses a method engineering (ME) approach. Since the OPF repository has already been enhanced by fragments from several other AO methodologies, we expect a "convergence to completion" (or near-completion) such that most of the PASSI fragments are likely to map to existing OPF fragments. Indeed, only seven new fragments (six of which are novel diagram types) are identified in this study.

1 Introduction: Acquisition of New Method Fragments

Method engineering (ME) offers a novel approach to a formalized way of creating a software development methodology [1-7]. Rather than create a single methodology in which there is significant intertwining of elements of the methodology, method engineering proposes that a methodology can be decomposed into a number of method fragments [5] (or method chunks). With the necessary interfaces on these method fragments, they can then be used in more than one methodology construction effort [7] and thus fulfil the criterion of methodological reuse [6]. Either this decomposition can be done on existing methodologies in order to extract these reusable method fragments or else method fragments can be identified *ab initio* (called Ad-Hoc construction in [6]). We apply the first of these approaches (decomposition of an existing methodology) to a case study of the PASSI agent-oriented methodology [8-10]. To guide the decomposition, we utilize an existing metamodel-underpinned repository of method fragments – the OPEN Process Framework (OPF) [11]. Within that framework, once a candidate method fragment for inclusion in the OPF repository has been identified (from PASSI), a decision can be made as to whether (1) to reject the proposal, (2) to accept as new fragment either "as is" (or with possibly small modifications to ensure compatibility with existing fragments) or (3) to merge the new fragment with others already in the repository, e.g. by taking an existing fragment and extending it to encompass the new detail.

M. Kolp et al. (Eds.): AOIS 2005, LNAI 3529, pp. 95 – 110, 2006.

The analysis of PASSI discussed here is one in a series of such extractions of method fragments from extant AO methodologies. It is therefore anticipated that the proposed additions of these newly identified method fragments to the OPF's repository will lead asymptotically to completeness such that the new method fragments likely to be identified will be few. In the next phase of the project, we intend to test out this hypothesis (that completion has been attained) by use of an external (methodological) data set.

In Section 2, we give a brief overview of both PASSI and the OPF, followed, in Section 3, by identification of appropriate method fragments from PASSI. For each fragment, we then ask whether it already exists in the OPF repository – if so, it will likely be rejected (decision 1) – or whether it should be accepted either as a new fragment (decision 2) or whether additional work is needed to merge together the newly proposed fragment with a pre-existing one (decision 3).

2 Very Brief Overviews of PASSI and OPF

2.1 PASSI

PASSI (A *P*rocess for *A*gent *S*ocieties *S*pecification and *I*mplementation) [8-10] offers a step-by-step requirement-to-code process for the development of an MAS (Fig. 1), integrating models and concepts from both the object-oriented (OO) software engineering and the agent-oriented paradigms. The methodology adopts (with minor extensions) the UML notation for its work products and targets the FIPA implementation environment.

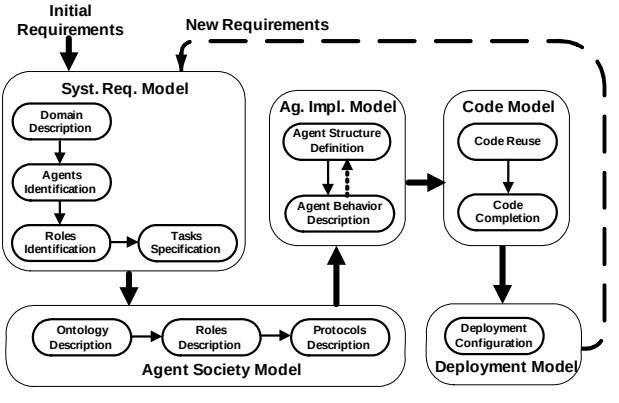

Fig. 1. Overview of PASSI

As depicted in Fig. 1, PASSI supports five phases of software development: (i) system requirements, to produce a use-case based description of the functionalities and an initial decomposition according to the agent paradigm; (ii) agent society, an analysis phase aimed at composing a model of domain ontology, social interactions and dependencies among the agents; (iii) agent implementation, which is a design

phase aimed at modelling the solution architecture in terms of required agents, classes and methods, composed of both a structural definition and a behavioural description of the whole system; (iv) code, the implementation phase aimed at modelling a solution at the code level - largely supported by pattern reuse and automatic code generation; (v) deployment, aimed at modelling the distribution of the system parts across a distributed platform. PASSI also includes support for testing, divided into two different stages: the agent test, where each individual agent is tested after its implementation, and the society test, where the multi-agent system is tested after its deployment.

The methodology is supported by PTK (PASSI Toolkit), a Rational Rose plug-in, and also by a repository of patterns for agents [12]. These tools proved very useful in the design and development of our systems because of the relevant level of automation they introduce in the process. This is particularly effective when entire portions of the model are reused from the patterns repository; this operation, that according to the PASSI prescription is performed during the design phase, also affects the coding activity since a significant portion of code is automatically generated starting from the pattern structure [13].

2.2 OPF

OPEN (Object-oriented Process, Environment and Notation) [11] is an established approach for developing software, primarily, but not exclusively, that with an object-oriented implementation. Within the OPEN approach, the most relevant element is the OPF (Fig. 2), which comprises a metamodel that defines all the methodology[1] elements at a high level of abstraction plus a repository that contains instances of those metalevels concepts supplemented by a set of construction guidelines.

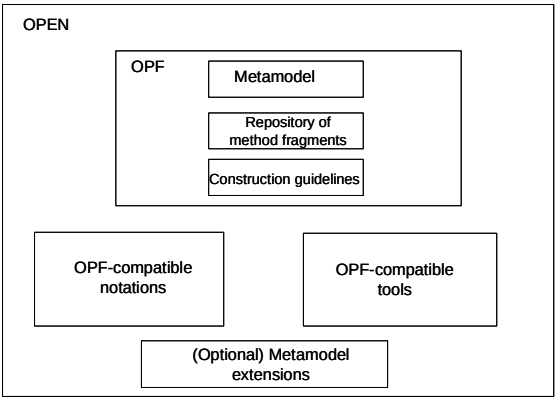

Fig. 2. The OPF consists of a metamodel, a repository and construction guidelines. OPEN consists of the OPF, OPF-compatible notations and tools and optionally metamodel extensions.

Each element in the repository is a *method fragment* generated, by instantiation, from the metamodel. There are several (meta)classes in the metamodel [11] but the

[1] We use a definition in which the term methodology encompasses both process and product [4].

most relevant for our study are two subclasses of Work Unit (namely Task and Technique) and the class Work Product.

Method engineering is then applied in the sense of identification of appropriate method fragments from the repository and their assembly into a full methodology. Using guidelines, such as ensuring that all output work products (except the deliverable code) are used elsewhere in the constructed method as inputs to some other task, a usable and quality methodology can be constructed for application on a specific project or organization – so called situational method engineering [1, 2, 14, 15]. Exemplar constructed processes can be found in [7].

3 Method Fragments in PASSI

In this section, we analyse PASSI by decomposing it (as an existing methodology) into fragments for process (cycles, phases), work units (tasks and techniques) and work products (models and diagrams). Each of these is first identified from PASSI and then we evaluate whether the pre-existing support in the OPF repository is adequate.

3.1 Fragments for Process Elements

3.1.1 Cycle

PASSI adopts an iterative and recursive lifecycle, where iteration is driven by new requirements, dependencies between structural and behavioural modelling, and dependencies between multi-agent and single-agent views. This lifecycle fits well into OPEN's "*Iterative, Incremental, Parallel Lifecycle*".

3.1.2 Phases

PASSI uses the term "phase" to refer to each of its steps in the MAS development process. However, in OPEN, the term "*Phase*" is defined as a *large-grained* span of time within the lifecycle that works at a given level of abstraction. Thus, "phases" of PASSI do not match the definition of OPEN "Phases", but instead correspond to OPF "*Tasks*", which are small-grained, atomic units of work that specify what must be done in order to achieve some stated result. We thus discuss PASSI's "phases" in Section 3.2 and note here that PASSI covers the OPF Phases of "*Initiation*" and "*Construction*".

3.2 Fragments for Tasks

In this section, we briefly describe each task fragment gleaned from PASSI and identify those that already exist in the OPF repository (decision 1), those that need to be added (decision 2) and those that enhance existing fragments (decision 3). In some instances, there is a one to many or many to one mapping (Table 1) as a consequence of the different granularities between the PASSI fragment and the OPF repository fragment.

3.2.1 "Domain Description"

This task aims to elicit the functional requirements of the target system via the development of use case diagrams (called Domain Description Diagrams in PASSI). It is a large task supported by three existing OPF tasks - as documented in Table 1. (Decision 1 fragment).

3.2.2 "Agent Identification"

PASSI identifies agents early in the development process because it views an MAS as a society of intended and existing agents. Agents are introduced from the identified requirements, and modelled in an Agent Identification Diagram(s) (see Section 3.4). Existing support from the OPF repository is shown in Table 1. (Decision 1 fragment).

Table 1. Mappings of PASSI fragments to existing OPF fragments

PASSI fragment	**Existing OPF fragment(s)**
Domain description	Elicit requirements Analyse requirements Use case modelling
Agent identification	Construct the agent model [16-19]
Role identification	Model agent's roles [20]
Task specification	Construct the agent model Model agents' tasks (new here)
Ontology description	Define ontologies [21] Construct the agent model
Role description	Model agents' roles [19]
Protocol description	Determine agent interaction protocol Determine agent communication protocol
Agents structure definition	Construct the agent model Model agents' tasks (new here)
Agents behaviour description	Construct the agent model Model agents' tasks (new here)
Code reuse	Code Identify appropriate reusable work products Acquire reusable work products Manage library of reusable components.
Code completion	Code
Deployment configuration	Create a system architecture

3.2.3 "Role Identification"

This task is concerned with the definition of agents' externally visible behaviour in the form of roles. Role identification produces a set of sequence diagrams (referred to as Role Identification Diagrams) that describe the scenarios in which the agents interact to achieve the required behaviour of the target system, and the roles played by each agent in these scenarios. Existing support from the OPF repository is shown in Table 1. (Decision 1 fragment).

3.2.4 "Task Specification"

This task is concerned with the definition of each agent's behaviour in the form of agent tasks. A Task Specification Diagram summarizes what each agent is capable of doing, ignoring information about roles that the agent plays when performing particular tasks. Existing support from the OPF repository is shown in Table 1. (Decision 1 fragment).

Support from OPF repository. The OPF Task "*Construct the Agent Model*" covers the specification of tasks or responsibilities for each agent. However, to make explicit PASSI's "task specification", we propose here a new Sub-Task to this Task, the new subtask to be named "Model agents' tasks". (Decision 2 fragment).

SUBTASK NAME: Model agents' tasks

Focus: Delineation of responsibilities/services of agents
Typical supportive techniques: "Responsibility identification", "Service identification", "Commitment management", "Deliberative reasoning", "Reactive reasoning", "Task selection by agents"

Explanation: This sub-task defines the tasks (or responsibilities or services) of each agent in the Agent Model. The internal structure of the tasks should be specified, i.e. the required knowledge and the involved operations/methods. Transitions among tasks within and between agents should also be defined. Task transitions are typically caused by events (e.g. an incoming message or task conclusion) or method invocation.

3.2.5 "Ontology Description"

This PASSI task develops domain-specific ontology for the target MAS in order to describe the pieces of domain knowledge that are ascribed to the agents. It produces two diagrams: Domain Ontology Description Diagram (to model the content of the ontology) and Communication Ontology Description Diagram (to model the agents' knowledge and the ontology used for each inter-agent communication). Existing support from the OPF repository is shown in Table 1. (Decision 1 fragment).

3.2.6 "Role Description"

This task provides an overview of the roles played by the agents, the changes in roles of an agent, the tasks performed by each role, the communications between roles, and inter-role dependencies. These elements are captured in Role Description Diagrams. Existing support from the OPF repository is shown in Table 1. (Decision 1 fragment).

3.2.7 "Protocol Description"

Each interaction protocol governing the inter-agent communications in the Communication Ontology Description Diagram (cf. PASSI task "*Ontology description*") needs to be documented using AUML sequence diagrams. Existing support from the OPF repository is shown in Table 1. (Decision 1 fragment).

3.2.8 "Agents Structure Definition"

This task specifies the general architecture of the system in terms of agents making up the system, their knowledge and their tasks, using a Multi-Agent Structure Definition Diagram. It also models the internal structure of each agent in terms of agent's knowledge and methods, and its tasks' knowledge and methods, using Single-Agent Structure Definition Diagrams. Existing support from the OPF repository is shown in Table 1. (Decision 1 fragment).

3.2.9 "Agents Behaviour Description"

This task influences and is influenced by the Agents Structure Definition task. At the system level, it specifies the transitions between the methods of different agents and/or the methods of different agents' tasks using Multi-Agent Behaviour Description Diagrams. At the agent level, it specifies the implementation of the methods of each agent and each agent's task via Single-Agent Behaviour Description Diagrams. Existing support from the OPF repository is shown in Table 1. (Decision 1 fragment).

3.2.10 "Code Reuse"

The designer should try to reuse predefined patterns [22] prior to the coding of agents and tasks. Existing support from the OPF repository is shown in Table 1. (Decision 1 fragment).

3.2.11 "Code Completion"

This is a conventional task in the system development process where the programmer completes the code of the application, taking as inputs the design specification and the reused patterns. Existing support from the OPF repository is shown in Table 1. (Decision 1 fragment).

3.2.12 "Deployment Configuration"

This task is particularly important if the system is highly distributed and/or contains mobile agents. A Deployment Configuration Diagram should be developed to detail the locations of agents. Existing support from the OPF repository is shown in Table 1. (Decision 1 fragment).

3.3 Fragments for Techniques

In this section, we briefly describe the techniques discussed in PASSI. These are not explicit so we have to identify appropriate technique fragments from the OPF repository or else identify areas where no such fragments pre-exist. Each subsection below refers to one of the PASSI tasks discussed above in Section 3.2.

3.3.1 For "Domain Description"

The functional requirements of the target system are described using a hierarchical series of use case diagrams, with the uppermost diagram serving as a context diagram.

Support from OPF repository. The OPF repository offers Technique "*Scenario development*" that directly supports the identification and construction of use cases and scenarios.

3.3.2 For "Agent Identification"

Starting from a sufficiently detailed use case diagram, agents are identified as a use case or a package of use cases. The functionality of the (package of) use case defines the functionality of the agent.

Support from OPF repository. Currently the OPF repository provides a Technique "*Intelligent agent identification*", which addresses the need for agents and agent modelling notation.

3.3.3 For "Role Identification"

Roles of each agent are identified by exploring all the communication paths between agents in the Agent Identification Diagram (produced by PASSI task "*Agent Identification"*). A communication path is captured as a «communicate» relationship between two agents in the diagram. At least one scenario should be developed for each path to specify how the agents interact, and to discover which role each agent plays during this interaction.

Support from OPF repository. The development of scenarios during the process of role identification is supported by OPF Technique "*Scenario development*". OPF Technique "*Collaboration analysis*" may also be useful to analyse inter-agent interactions for role discovery.

3.3.4 For "Task Specification"

The designer should examine all Role Identification Diagrams produced by task "*Role Identification*" (i.e. all scenarios that the agents participate). From each Role Identification Diagram (i.e. each scenario), a collection of related tasks can be identified for each agent by exploring the interactions and the internal actions that the agent performs to accomplish the scenario's purpose. Grouping all the tasks identified for a particular agent will result in a Task Specification Diagram for that agent.

Support from OPF repository. The identification of agents' tasks can be assisted by various OPF Techniques such as "*Responsibility identification*", "*Service identification*", "*Commitment management*", "*Deliberative reasoning*", "*Reactive reasoning*" and "*Task selection by agents*" [19].

3.3.5 For "Ontology Description"

PASSI does not offer any techniques for the development of the Domain Ontology Description Diagram, such as how to identify the concepts, predicates, actions and relationships in the ontology. Regarding the Communication Ontology Description Diagram, agents in the diagram are those identified by the Agent Identification Diagram, while the communications between agents are deduced from the interactions between agents' roles in Role Identification Diagrams. The designer must define agents' knowledge (represented as attributes) and the ontology governing each inter-agent communication in terms of the elements of the Domain Ontology Description Diagram.

Support from OPF repository. For the specification of domain ontology, OPF Technique "*Domain analysis*" can be applied to identify the relevant domain-specific concepts, predicates, actions and their relationships. Regarding the specification of agents' knowledge in terms of domain ontology, OPF Technique "*Agent Internal Design*" [16] needs to be enhanced in order to exercise the consistency rule between the definition of agents' knowledge and the definition of domain ontology. OPF Technique "*Interaction modelling*" is also useful here.

3.3.6 For "Role Description"

The roles of each agent are identified from the Role Identification Diagram. Communications between roles can be deduced from the communications between agents in

Communication Ontology Description Diagram, using exactly the same names for the communication relationships. Changes in roles of an agent and inter-role dependencies should also be specified. Three potential types of dependencies are:

- Service dependency: where a role depends on another role to bring about a goal;
- Resource dependency: where a role depends on another for the availability of an entity; and
- Soft-Service or Soft-Resource dependency: where the requested service or resource is helpful but not essential to bring about a role's goal.

PASSI does not document any techniques for the identification of tasks for each agent's role.

Support from OPF repository. Support for modelling communication between roles, changes in roles of an agent and inter-role dependencies can be accommodated by OPF Technique "*Role Modelling*", although this technique is to be enhanced here by inclusion of the various guidances suggested by PASSI. For the identification of tasks for each role, OPF Techniques "*Responsibility identification*", "*Service identification*" and "*Scenario development*" should be applied.

3.3.7 For "Protocol Description"

PASSI advocates the adoption of standard FIPA interaction protocols and AUML sequence diagrams to document these protocols. If the existing FIPA protocols are found inadequate for the target system, the designer may specify his or her own, using the same FIPA documentation's approach.

Support from OPF repository. Conventional OPF Technique "*Interaction modelling"* and OPF Techniques "*Contract net*", "*Market mechanisms*" and "*FIPA-KIF compliant language*" [19] can be applied to specify protocols and the exchanged messages between agents.

3.3.8 For "Agents Structure Definition"

The names of the agents in the Multi-Agent Structure Definition Diagram can be derived from the Agent Identification Diagram, their knowledge from Communication Ontology Description Diagram, their tasks from Task Specification Diagrams and their communications from Role Description Diagrams. The internal structure of each agent should then be defined in a Single-Agent Structure Definition Diagram (one diagram for each agent). The agent internal structure consists of the agent's knowledge and methods, together with the knowledge and methods of each of its tasks. The designer should not overlook methods that are needed for the implementation platform, e.g. constructor and shutdown methods. Tasks that require inter-agent communication should also contain methods that deal with communication events.

Support from OPF repository. The Technique of *"Organizational structure specification"* [17] is useful in multi-agent structure definition; while the specification of agent internal structure (including agent knowledge, tasks, methods etc) is directly supported by OPF Technique "*Agent internal design*" [16]. In addition, since PASSI employs the OO concepts of class, attribute and method to model agents and agents' tasks, the OPF conventional Technique "*Class internal design*" is also appropriate.

3.3.9 For "Agents Behaviour Description"

One or more Multi-Agent Behaviour Description Diagrams should be developed for the target system to show the transitions between the methods of agents and/or methods of agent's tasks. These transitions represent either events (e.g. an incoming message or task conclusion) or invocation of methods. They can be identified from inter-role/inter-agent communications captured in the Role Identification Diagram, Task Specification Diagram and Communication Ontology Description Diagram. If the transition represents an exchanged message, the message's performatives must be consistent with the protocol defined in the Communication Ontology Description Diagram and Role Description Diagram, and the message's content should contain elements defined in the Domain Ontology Description Diagram. With regard to the implementation of methods (of agent classes and task classes), standard OO diagrams such as flowcharts and state diagrams can be used as Single-Agent Behaviour Description Diagrams.

Support from OPF repository. Standard OPF Techniques "*Event modelling*" and "*State modelling*" are appropriate to the identification and modelling of transitions between methods and implementation of each method (no matter whether the methods belong to agents or to agents' tasks).

3.3.10 For "Code Reuse"

Code reuse does not merely mean the reuse of pieces of codes, but also pieces of design of agents and tasks. The designer should thus look at the design diagrams detailing the library of patterns rather than at the code directly. PASSI provides an add-in to the Rational Rose UML CASE tool (called "PASSI Toolkit") and a pattern reuse application (called "Agent Factory") that assist in code reuse. "PASSI Toolkit" (PTK) can generate the code for all skeletons of agents. In the context of the generation of PASSI from the newly enhanced OPF repository (as described here), PASSI tools become elements of the OPF-compatible tools (Fig. 2).

Support from OPF repository. The OPF repository provides various Techniques for reuse that can be applied to PASSI, namely "*Pattern recognition*", "*Library class incorporation*", "*Library management*" and "*Reuse measurement*".

3.3.11 For "Code Completion"

No specific techniques are documented by PASSI because this is a classical task of the programmer.

Support from OPF repository. The OPF repository contains a number of Techniques for coding, which, although originally intended for objects, are equally applicable to agents, e.g. "*Inspection*", "*Creation charts*", "*Pair programming*", "*Screen scraping*" and "*Wrappers*".

3.3.12 For "Deployment Configuration"

No techniques are given by PASSI to support agent deployment configuration, for example how to allocate agents to processing units or how to configure agent mobility.

Support from OPF repository. The OPF repository offers Technique "*Distributed systems partitioning and allocation*". However, it offers inadequate support for the

deployment configuration of agent systems, including mobility of agents. Since PASSI offers no guidance here, in the context of this paper, we must defer this extension to future work.

3.3.13 Summary

Although only a single subtask is identified as needing adding to the OPF repository (together with the need to investigate extending a single technique (*Distributed systems partitioning and allocation*), this does not reflect upon any lack of comprehensiveness in PASSI itself. The reason is that a significant number of other agent-oriented methodologies have already been analysed [20], each of which has provided method fragments that could equally well have been derived from PASSI. We have chosen not to highlight these here to avoid duplication with those previously published [16-19, 21].

3.4 Fragments for Work Products

All work products of PASSI are represented in UML notation although with some small extensions.

3.4.1 System Requirements Model

This is an anthropomorphic model of the system requirements in terms of agency and purpose. It is composed of the following types of diagrams:

- *Domain Description Diagram*: This is a standard UML use case diagram that is used (by PASSI task "*Domain Description*") to capture the functional description of the target system.

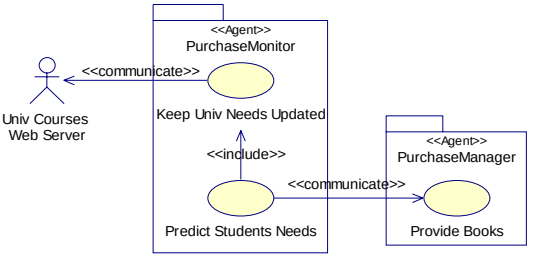

Fig. 3. Agent Identification Diagram

- *Agent Identification Diagram*: One or more use cases in the above use case diagrams are grouped into stereotyped packages to form Agent Identification Diagrams (Fig. 3). This assumes that use cases are fully contained in a single agent, which is not the case for object-oriented systems. The names of the packages are the names of the resulting agents. Relationships between use cases of different agents are stereotyped as «communicate», while relationships between uses cases of the same agent are modelled using the standard UML relations (i.e. «include» and «extend»). This is a new style of diagram recommended for addition to the OPF repository.

- *Role Identification Diagram*: This is a UML sequence diagram where objects represent agent roles, specified using the syntax <role-name>:<agent_name>. An agent may play distinct roles within the same sequence diagram. Messages in the sequence diagram may either signify events generated by the environment or communication between roles. This is a new style of diagram recommended for addition to the OPF repository.
- *Task Specification Diagram*: This diagram is drawn as a UML activity diagram with two swimlanes. The right-hand lane contains a collection of tasks of the target agent, while the left-hand lane specifies the relevant tasks of other interacting agents. Relationships between tasks signify transitions between them (e.g. exchanged messages or task triggering events). This is a new style of diagram recommended for addition to the OPF repository.

3.4.2 Agent Society Model

This model captures the communications and dependencies among agents in the target system. It is composed of the following types of diagrams:

- *Domain Ontology Description Diagram*: This diagram models the domain ontology of the target system in terms of concepts (domain entities), predicates (assertions on properties of concepts), actions (performed in the domain) and their relationships (association, generalization and aggregation). This diagram is represented as a UML class diagram, while the elements of the ontology (i.e. concepts, predicates, actions and relationships) are described in an XML schema.
- *Communication Ontology Description Diagram*: This is a UML class diagram that shows all agents of the system, their knowledge (represented as attributes) and the ontology governing their communications (Fig. 4). Each communication (drawn from the initiator to the participant) is characterized by three attributes: ontology, language and interaction protocol, which are grouped into an association class. Roles played by agents are denoted at the respective ends of the association lines. This is a new style of diagram recommended for addition to the OPF repository.

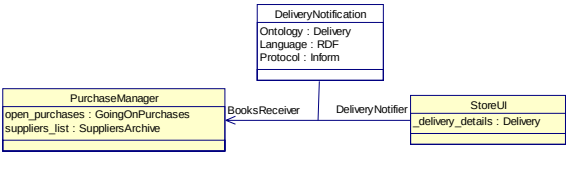

Fig. 4. Communication Ontology Description Diagram

- *Role Description Diagram:* This is a UML class diagram that shows agents as packages, and agents' roles as classes (Fig. 5). Each role's tasks are specified in the operation compartment of the role class. Connections between roles represent either changes of roles (if the roles belong to the same agent) or inter-role communications (if the roles belong to different agents). Dependencies among roles are also shown. This is a new style of diagram recommended for addition to the OPF repository.

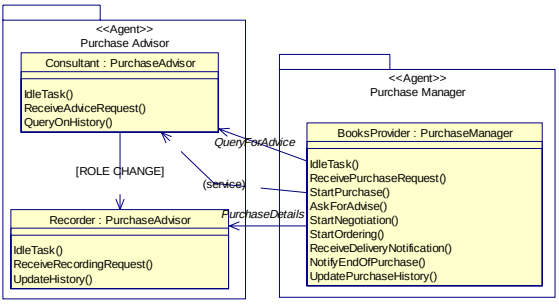

Fig. 5. Role Description Diagram

- *AUML Sequence Diagram*: This diagram is used for documenting inter-agent interaction protocols.

3.4.3 Agent Implementation Model

This model captures the solution for the target MAS in terms of classes and methods. It consists of four types of diagram:

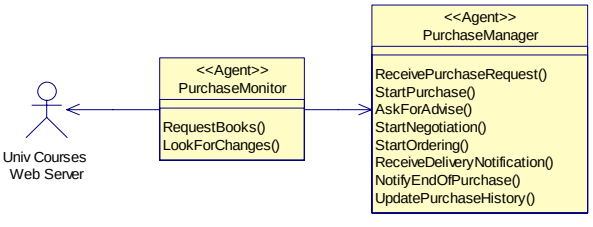

Fig. 6. Multi-Agent Structure Definition Diagram

- *Multi-Agent Structure Definition Diagram*: This is a UML class diagram where classes represent agents and associations between classes signify inter-agent communications (Fig. 6). Attributes represent the agents' knowledge, while operations are used to specify agents' tasks. This is a new style of diagram recommended for addition to the OPF repository.
- *Single-Agent Structure Definition Diagram*: One UML class diagram is developed for each agent. This diagram contains one main class to represent the target agent, and multiple inner classes to represent the agent's tasks (one inner class for each task). The knowledge and methods of each agent class and task class should be specified in the attribute and operation compartments respectively.
- *Multi-Agent Behaviour Description Diagram*: This is a UML activity diagram where each swimlane specifies the methods of each agent or each agent's task. The methods (represented as activities) are connected with each other through transitions, i.e. events (e.g. an incoming message or a task conclusion) or invocations of methods.

- *Single-Agent Behaviour Description Diagram*: This diagram can be represented as standard UML flowcharts, state diagrams or even semi-formal text description.

3.4.4 Code Model

This model captures the codes for implementing the solution.

3.4.5 Deployment Model

This model contains a Deployment Configuration Diagram, which is represented as a UML deployment diagram. This diagram shows the locations of agents (i.e. the implementation platforms and processing units where the agents reside), the agents' movements and their communication. A «move_to» stereotyped connection is introduced by PASSI to model agent mobility, connecting an agent from its initial processing unit to the final location.

3.4.6 Recommendations

PASSI focusses on the use of UML diagrams. There are, however, some interesting observations to make. Firstly, in the UML there is a tendency to have a one to one relationship between a diagram type and its context of application. In PASSI (and also incidentally in Tropos e.g. [23]), one diagram type is used to serve many purpose. In PASSI, the UML class diagram, for example, is used as (i) a domain ontology description diagram, (ii) a communication ontology description diagram, (iii) a role description diagram, (iv) a multi-agent structure definition diagram and (v) a single-agent structure definition diagram. Such multi-viewpoint usage can be beneficial in terms of only using one notational style but potentially confusing unless the boundaries between the diagram types and the contexts and scales of the various viewpoints are carefully delineated.

Table 2. New Work Products, derived from PASSI, recommended for inclusion in the OPF repository

Agent Identification Diagram
Communication Ontology Description Diagram
Multi-Agent Structure Definition Diagram
Role Description Diagram
Role Identification Diagram
Task Specification Diagram

In terms of OPF method fragments, six of the PASSI diagrams are distinctive to warrant proposal for inclusion in the OPF repository (Table 2).

Inadequate support for distributed systems partitioning and allocation was identified in both PASSI and the OPF and remains a topic for future investigation.

4 Conclusion

By decomposing PASSI into a set of fragments and then comparing these newly derived fragments with those already stored in the OPF repository, as enhanced by

previous AO methodology studies [16-21], we have identified only one major WorkUnit fragment (Subtask: Model agent's tasks) that needs to be added to this particular repository plus a recommendation to (a) extend the "Distributed systems partitioning and allocation" technique described in [24] and (b) consider six of the twelve PASSI work products for inclusion in the repository. The next stage of the work will posit the hypothesis that completeness of the repository has been reached, testing this by means of an external data set, as provided in [25]. It is also interesting to note that the work reported here represents an evaluation of the possibilities of interaction of the FIPA Methodology TC[2] approach with the OPF one. Since the original PASSI fragments have been built by following the FIPA Method Fragment Specification and their introduction in the OPF repository has been smooth enough, we think there is a reasonable hope of making the two approaches converge towards some interoperability level and we plan to explore this possibility further.

References

1. Kumar, K., Welke, R.J.: Method engineering: a proposal for situation-specific methodology construction, in *Systems Analysis and Design: A Research Agenda,* (eds. W.W. Cotterman and J.A. Senn), J. Wiley & Sons, NY, USA (1992) 257-269.
2. Brinkkemper, S.: Method engineering: engineering of information systems development methods and tools, *Inf. Software Technol.*, **38(4)** (1996) 275-280.
3. Ralyté, J., Rolland, C.: An assembly process model for method engineering, in K.R. Dittrich, A. Geppert and M.C. Norrie (Eds.) *Advanced Information Systems Engineering*, LNCS2068, Springer-Verlag, Berlin (2001) 267-283.
4. Rolland, C., Prakash, N., Benjamen, A.: A multi-model view of process modelling, *Req. Eng. J.*, **4(4)** (1999) 169-187
5. van Slooten, K., Hodes, B.: Characterizing IS development projects, in S. Brinkkemper, K. Lyytinen and R. Welke (Eds.) *Procs. IFIP TC8 Working Conference on Method Engineering: Principles of method construction and tool support,* Chapman & Hall, London (1996) 29-44.
6. Ralyté, J.: Towards situational methods for information systems development: engineering reusable method chunks, *Procs. 13th Int. Conf. on Information Systems Development. Advances in Theory, Practice and Education* (eds. O. Vasilecas, A. Caplinskas, W. Wojtkowski, W.G. Wojtkowski, J. Zupancic and S. Wrycza), Vilnius Gediminas Technical University, Vilnius, Lithuania (2004) 271-282.
7. Henderson-Sellers, B., Serour, M., McBride, T., Gonzalez-Perez, C., Dagher, L.: Process construction and customization, *J. Universal Computer Science*, **10(4)** (2004) 326-358
8. Burrafato, P., Cossentino, M.: Designing a multi-agent solution for a bookstore with the PASSI methodology. *Procs. 4th International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2002), Toronto* (2002)
9. Cossentino, M.: From requirements to code with the PASSI methodology, in *Agent-Oriented Methodologies* (eds. B. Henderson-Sellers and P.Giorgini), Idea Group (2005) 79-106.
10. PASSI website. http://www.pa.icar.cnr.it/passi/
11. Firesmith, D.G., Henderson-Sellers, B.: *The OPEN Process Framework,* Addison-Wesley, UK (2002)
12. Chella, A., Cossentino, M., Sabatucci, L.: Tools and patterns in designing multi-agent systems with Passi. *WSEAS Transactions on Communications*, **3(1)** (2004) 352-358.

[2] http://www.pa.icar.cnr.it/~cossentino/FIPAmeth/

13. Cossentino, M., Sabatucci, L., Chella. A.: Patterns reuse in the PASSI methodology. In Engineering Societies in the Agents World IV, 4th International Workshop, ESAW 2003, Revised Selected and Invited Papers, volume XIII of Lecture Notes in Artificial Intelligence. Springer-Verlag (2004)
14. Welke, R., Kumar, K.: Method engineering: a proposal for situation-specific methodology construction, **in** *Systems Analysis and Design: A Research Agenda* (Cotterman and Senn, eds.), Wiley (1991)
15. Odell, J.J.: Keynote paper: a primer to method engineering, **in** *Method Engineering. Principles of method construction and tool support* (eds. S. Brinkkemper, K. Lyytinen and R.J. Welke), Chapman & Hall (1996) 1-7
16. Tran, Q.N., Henderson-Sellers, B., Debenham, J.: Incorporating the elements of the MASE methodology into Agent OPEN. *Procs. 6th Int. Conference on Enterprise Information Systems (ICEIS'2004)* (2004) 380-388.
17. Henderson-Sellers, B., Debenham, J., Tran, Q.N.: Adding Agent-Oriented Concepts Derived from GAIA to Agent OPEN. *Advanced Information Systems Engineering. 16th International Conference, CAiSE 2004, Riga, Latvia, June 2004 Proceeding* (eds. A. Persson and J. Stirna), LNCS 3084, Springer-Verlag, Berlin (2004) 98-111.
18. Henderson-Sellers, B., Tran, Q-N.N., Debenham, J.: Incorporating elements from the Prometheus agent-oriented methodology in the OPEN Process Framework. *Procs. AOIS@CAiSE*04*, Faculty of Computer Science and Information, Riga Technical University, Latvia (2004) 370-385
19. Henderson-Sellers, B., Debenham, J.: Towards OPEN methodological support for agent-oriented systems development. *Procs. 1st International Conference on Agent-Based Technologies and Systems* (eds. B.H. far, S. Rochefort and M. Moussavi), University of Calgary, Canada (2003) 14-24.
20. Henderson-Sellers, B.: Creating a comprehensive agent-oriented methodology - using method engineering and the OPEN metamodel, in *Agent-Oriented Methodologies* (eds. B. Henderson-Sellers and P. Giorgini), Idea Group (2005) 368-397.
21. Henderson-Sellers, B., Tran, Q-N.N., Debenham, J., Gonzalez-Perez, C.: Agent-oriented information systems development using OPEN and the Agent Factory. *Information Systems Development Advances in Theory, Practice and Education, 13th International Conference on Information Systems Development, ISD 2004, Vilnius, Lithuania, September 2004, Proceedings*, Kluwer, New York, USA (2050) 149-160.
22. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley (1994)
23. Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopolous, J., Perini, A.: Tropos: an agent-oriented software development methodology, *Autonomous Agents and Multi-Agent Systems*, **8(3)** (2004) 203-236.
24. Henderson-Sellers, B., Simons, A.J.H., Younessi, H.: *The OPEN Toolbox of Techniques*, Addison-Wesley, UK (1998)
25. Zhang, T.I., Kendall, E., Jiang, H.: An agent-oriented software engineering methodology with applications of information gather systems for LLC, *Procs AOIS-2002*, (eds. P. Giorgini, Y. Lespérance, G. Wagner and E. Yu), Toronto (2002) 32-46

Foundations of Ontology-Based MAS Methodologies

G. Beydoun[1], N. Tran[2], G. Low[2], and B. Henderson-Sellers[3]

[1] School of Economics and Information Systems, University of Wollongong, Australia
beydoun@uow.edu.au
[2] School of Information Management and Technology Management, University of New South Wales, Australia
{g.low, numitran}@unsw.edu.au
[3] Faculty of Information Technology, University of Technology, Australia
brian@it.uts.edu.au

Abstract. Support for software extensibility, interoperability and reuse are critical concerns for long term commercial viability of any MAS and they underpin the eventual adoption of agent technology by industry. Existing AOSE methodologies lack adequate support for these concerns. We argue in this paper that a methodology that uses ontologies as a central modelling artifact, beyond the analysis phase, is better equipped to address those concerns.

We observe that the influence of ontologies in Knowledge-based Systems (KBS) methodologies extended well beyond the initial analysis phase, leading to domain-independent KBS methodologies in the '90s. We reflect on those lessons and on the roles of ontologies in KBS development. We analyse and identify which of those roles can be transferred to an ontology-based MAS development methodology. We identify ontology-related inter-dependencies between the analysis and design phases. We produce a set of recommendations towards creating a domain-independent MAS methodology that incorporates ontologies beyond the analysis phase. We identify the essential features and sketch the characteristic tasks within both the analysis and design phases.

1 Introduction

AOSE researchers have proposed a number of methodologies to support the analysis and design of MASs e.g. [1-3]. Our research [4] has revealed that existing AOSE methodologies lack support for software extensibility (extending the functionality and lifetime of an MAS), software interoperability with other systems in heterogenous environments and reuse of modelling effort (e.g. as requirements change). This paper argues that a methodology that is ontology based (that is, it uses ontologies as a central model beyond the analysis phase) can better produce reusable MAS designs and components (beyond the ontologies themselves). In this paper, we discuss the software engineering requirements to create such an ontology-based MAS methodology.

This paper describes the first step towards the long term reuse of software engineering knowledge and effort involved in developing MASs. The long term reuse may take the form of extending functionality of an existing system, reusing components of an existing system in an entirely different context or to creating a new system using the design (in whole or parts) of an existing system. This is akin to the type of reuse in KBS that allowed profiting from knowledge engineering efforts by extending the

M. Kolp et al. (Eds.): AOIS 2005, LNAI 3529, pp. 111 – 123, 2006.

functionality and lifetime of a given system as a whole or in part. This was based on interoperability between various KBS components and reuse of original designs as requirements change. We argue that these concerns will need to be addressed at design time of an MAS and accommodated in agent methodologies. We use as a guide the roles of ontologies of reuse and domain-independent development in modern Knowledge-based Systems (KBSs) rooted in the situated view of knowledge (as advocated for example in [5]). This leads us to highlight the often-overlooked ontology-related interactions between the analysis and design software development phases for MASs (with intelligent knowledge-based agents as advocated by the popular BDI agent model [6]).

Tran *et al.* (2003) show that ontologies are not used in the design process of MASs in the more prominent of the Agent-Oriented Methodologies. Using the domain independence of KBS methodologies as a guide, we believe that what is required so that ontologies are effectively accommodated as ready components in MAS architectures and throughout the design phase is a domain-independent methodological approach founded on ontological analysis throughout the whole software development lifecycle. This paper proposes using ontologies in both the analysis and design phases. We analyse the interplay between analysis and design and describe the software engineering requirements of an ontology-based domain-independent methodology.

This paper is organized as follows: Section 2 presents the background and related work. In particular, it highlights how ontologies originated in the process of developing knowledge-based systems to address reuse concern. This section exposes how these original motives of using ontologies are overlooked in MAS development processes. Section 3 views a multi-agent system as a distributed collection of knowledge-based systems and accordingly highlights how ontologies can be accommodated during and throughout the development of an MAS. Section 4 uses the analysis of Section 3 to sketch an actual MAS ontology-based methodology to accommodate reuse of MAS components and software engineering products. Section 5 concludes with a summary and future work.

2 Background and Related Work

Ontologies are an explicit formal specification of a shared conceptualization [7]. They have been employed in many computing areas e.g. knowledge management [8], natural language processing [9], information retrieval and integration [10]. In knowledge-based systems development, they have been successfully used to enhance reusability and interoperability and to verify various products of software development e.g. [11]. They have lately become an essential tool in sharing models of dependencies between webpages, in order to transform WWW into a semantic web [12]. In this paper, we propose using ontologies as a central software engineering construct throughout the whole development lifecycle of MAS, and address software interoperability, reusability and verification concerns for MAS. We focus on protecting the different facets of software engineering investments associated with using an MAS, not only through interoperability of systems and reuse of their components but also through the reuse of human skills acquired and designs generated during development. This will accelerate and spearhead the adoption of MAS technology into the mainstream software development community.

In reviewing the related work, we take two perspectives: we first focus on the traditional roles of ontologies in the ’80s and ’90s in enhancing the interoperability and reusability of knowledge-based systems components. We then examine the current role of ontologies in MAS and note how their traditional roles are essentially downplayed in current AOSE practices. The contrast between these two perspectives is later used as a basis to analyse how we can extend the roles of ontologies into MAS development.

2.1 Traditional Roles of Ontologies in Knowledge-Based Systems

In the ’80s and ’90s, developers of Knowledge-based Systems (KBS) were frustrated by two conflicting goals: the more *usable* and effective systems were for a given problem, the harder it was to adapt them to new problems. This became known as the usability-reusability tradeoff [13]. To address this, the idea of decoupling problem solving knowledge from domain knowledge was pursued. This idea underlies the use of ontologies in single agent systems. Reusability of system design is recognized as a key concern in single agent knowledge-based systems [11, 14] and is the impetus for the ontology-based architectural view of a KBS as being formed from two components: a PSM (Problem Solving Method) and a suitable ontology (Fig. 1). This view is central to many KBS methodologies e.g. [15-18]. It was the impetus for most of the KBS research in the ’80s and ’90s, with the aim of reducing KBS analysis and design to ontology engineering coupled with a suitable choice of a PSM from some existing library of PSMs [19]. Ontologies were used to support the reuse of PSMs in different problem areas. Alternatively, PSM components permitted reuse of ontologies to address different problems within the same area. The success of this view is expressed in KBS methodologies e.g. [15-18]. Delving further into history of knowledge-based systems, this view was born out of two decades of research in Artificial Intelligence (AI).

The era of knowledge-based systems and expert systems began in AI as a response to the failure of general problem solving approaches. Newell [20] suggested the notion of *knowledge level*, i.e. the idea of specifying all necessary knowledge that a system needs at a level that, roughly speaking, corresponds to the level at which humans communicate their knowledge to their fellows. This idea penetrated much of the work that followed in the ’80s and ’90s [19, 21, 22]. The idea of the *knowledge level* itself received some attention, resulting in developments such as a finer categorization of knowledge into domain knowledge, problem solving knowledge etc., each type of knowledge describing different aspects of what a human problem solver or expert may use. The idea of splitting a problem solving method from its application domain was born out of investigation of techniques necessary to use the provided knowledge and to turn it into a working system. In other words, it has been realized that problem solving generic techniques such as deductive reasoning techniques or forward and backward chaining suffice to utilize knowledge in order to address all sorts of problems (termed ‘weak problem solving methods’). Rather, specific techniques for different kinds of problems are necessary in order to build relatively complete and competent systems. This resulted in collections of problem-solving methods that are used in conjunction with domain ontologies together with the relevant domain knowledge (see, e.g. [19, 23, 24]). Ontologies provided domain-dependent reusable encapsulation of the structural basis of domain knowledge. They provide a structured representation of a knowledge area related to a problem to be solved that can be processed by a computer.

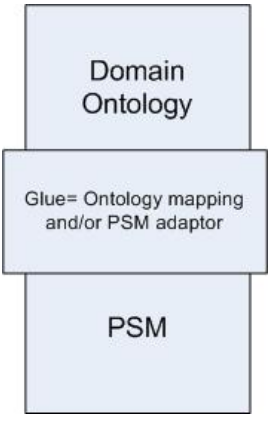

Fig. 1. KBS architecture based on an ontology and a Problem Solving Method

2.2 Ontologies and AOSE

As discussed above, the reliance on ontologies produced reusable and effective components for building robust KBS more economically. Moreover, this led to methodologies founded on ontological analysis and that are consequently domain-independent. Examples include KAMET [18], Ibrow3 [17], KAMET II [25], KADS [16] and CommonKADS [26]. In contrast, in the current state of the art in MAS Software Engineering (usually referred to as Agent Oriented Software Engineering (AOSE)), agent-oriented methodologies are being published at an increasing rate and most acknowledge their own suitability for a given class of applications. For example, Adelfe [27] targets adaptive systems. Passi [28] is limited to a class of communication architectures. Other notable examples are Gaia [3], Tropos [1] and Prometheus [2]. Few MAS methodologies include ontologies in their models and processes e.g. [29, 30]. The inclusion of ontologies in such works is confined to the analysis phase of the development. For instance, the authors of Balby [31] distinguish between an initial ontology and a domain model geared towards designing an MAS and they specify how a *domain model* that includes goal and role analyses is developed from an initial ontology. Similarly, in [30], the MaSE methodology is extended to incorporate the use of an ontology to mediate the transition between the goal and the task analyses (both are within the analysis phase). Our work in this paper is perhaps closest to recent work in [32], which recognizes the role of using ontologies for verification of models during the analysis phase. Outside the analysis phase, ontologies currently are mainly used to express a common terminology for agent interactions in an MAS e.g. [33]. These interactions have no parallel within a single agent KBS (since an agent does not usually need to interact with itself!). We find that the initial motivation for using ontologies (for single agent systems), that of enhancing reuse (cf. [7]) of system architectures and components, is absent in AOSE.

In the next section, we examine how assumptions about the way knowledge is used vary between a single agent KBS and individual agents in an MAS. We use these differences to formulate ways to enhance the use of ontologies for developing an MAS. We highlight the changes required by methodological approaches of MAS

development in order to accommodate reuse together with ontology-oriented MAS analysis and design phases.

3 Ontologies for MAS Development

The idea of using domain ontologies in KBS aims at reusing part of the domain knowledge in different systems. That is, a domain is characterized by a set of objects that are referred to by a set of terms that are deemed relevant and that can be used by different systems to handle different types of tasks. The development of reusable ontologies creates the problem that a general-purpose ontology is very rich, while for a particular task only a small part of it will actually be needed. To compensate, KBS developers carefully choose a suitable problem-solving method and adapt the ontology used to a suitable level of refinement. This idea does not have a direct parallel from single agents to multi-agents. Modern MAS methodologies still do not incorporate domain-independent ontological analysis in their processes. Viewing a Multi-Agent System as a 'distributed knowledge' based system, we explore in this section how we can incorporate domain-independent ontological analysis in an MAS methodology by considering differences in the way knowledge is used for an agent system and for an MAS.

In an MAS, each agent has a localized knowledge base. Agent knowledge bases may overlap. Shared knowledge is usually of some concern, but certainly each will have its own private knowledge component. In an MAS, two or more agents interact or work together to achieve a set of goals. The coordination between agents possessing diverse knowledge and problem-solving capabilities usually permits the achievement of global goals that cannot be otherwise achieved by a single agent working in isolation. MASs are thought to be an answer to a number of shortcomings of general problem-solving methods [5]: incomplete knowledge requirement specification, incomplete PSM requirement and limited computational resources. MASs are particularly useful in the engineering of open, dynamic and adaptive systems. Associated with these shortcomings that MASs address, we note the following differences between agents within an MAS and a single agent KBS: an MAS may have different PSMs for different agents, some agent ontologies may be incomplete in an MAS, individual PSMs for agents may be insufficient for their own goals in an MAS, and agents within an MAS may have limited execution resources. In what follows, we overview how each of these differences characterize the way agents may interact within an MAS, noting six additional requirements to support these differences. Later, in Section 4, we sketch an appropriate ontology-based methodology. In this discussion, we ignore mobility and any additional security requirements that the distributed and mobile code may impose.

3.1 Heterogeneity of PSMs in an MAS

In the case of multi-agent systems, different problem solvers operate on the same domain. Using ontologies in an MAS is complicated by having to provide knowledge requirements to different PSMs at the same time. Whilst individual PSM may operate at different levels of abstraction of the domain, they still need to share their results

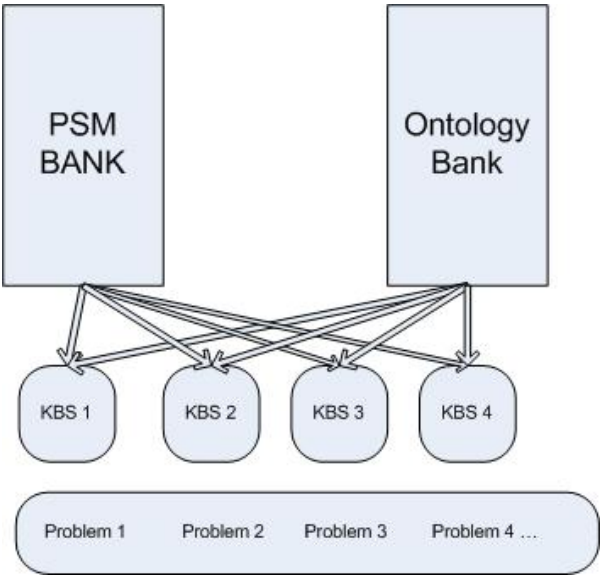

Fig. 2. As new problems arise, the PSM and the ontology banks are used to construct suitable KBSs. An ontology from the *ontology bank* strengthens a given PSM from the *PSM bank* to suit the domain.

using a common terminology. PSMs may be complementary and may have different degrees of strength. How much specificity they exhibit to a given domain may vary. In the commonly adopted Belief-Desire-Intention (BDI) [34] architecture of agents, a PSM essentially specifies how plans are generated and discarded and how beliefs are updated and maintained/shared. In contrast, within a single agent KBS, ontologies were conceived and used to strengthen a single PSM for a given domain. Their use for KBS was never intended to *simultaneously* strengthen different PSMs for the same domain (see Fig. 2). Therefore, in developing MASs, we may additionally need the following requirements:

Requirement 1: Ontology mappings are required to allow individual problem solvers (of individual agents) to interact and to use a common domain conceptualization.

Requirement 2: Verification of individual PSM knowledge requirements against allocated ontologies is required at design time.

3.2 Incompleteness of Individual Agent Ontologies in an MAS

A domain ontology underlying knowledge requirements of all agents is available. However, the version available to an individual agent, matching its PSM, is not necessarily complete (as is assumed to be the case for single agent systems). In addition to 1 and 2, we add:

Requirement 3: Knowledge extensibility is required at the agent level at least to accommodate any new ontological units added to the system about the domain. This can often create inconsistencies [35].

Requirement 4: Associated with 3, a structured and understood knowledge representation is required to resolve inconsistencies.

3.3 Incompleteness of PSMs in an MAS

An agent PSM is not assumed to be sufficiently powerful to respond to all events it encounters during its lifetime within an MAS. It usually negotiates cooperation from other agents. Current practices often assume that functional goal analysis is sufficient to specify the knowledge requirement for agents [1], and any deficiencies in its later problem-solving capacity are assumed to be offset by that cooperation. However, in our view, without consideration of its actual PSM (or other available PSM within the system), there is no guarantee that this cooperation would ultimately work. This suggests:

Requirement 5: Iteration between the PSM design and the goal analysis is required to ensure that the chosen problem solver for a given agent is capable of meeting its specified goals.

Requirement 6: A consideration of the total PSMs of all agents is required to ensure that system goals are achievable.

3.4 Limitation of MAS External Resources

Agents are limited by their resources e.g. computation, storage and response time. It is often assumed that agents cooperate through sharing of their processing resources. This requires synchronization. Common agent platforms such as Jade can resolve this.

Towards accommodating the above six ontology-related design considerations within an existing methodological framework, in the next section we develop the above analysis into software engineering (SE) requirements to improve current AOSE practices.

4 Ontologies for MAS Reuse

In this section, we refine the analysis of the previous section to sketch the key features of an ontology-based methodology. The sketched methodology is motivated by the original KBS drive for using ontologies (of Section 2) for *reuse.* Similar to KBS development, we assume that the choice of PSM may be made independently of domain analysis. Moreover, we assume that a domain ontology describing domain concepts and their relationships is available. Such an ontology may be available from an existing repository e.g. [36] or a domain analysis may be considered the first stage of developing the system. The purpose of such a domain analysis would only be to identify concepts and their relationships. Cordi *et al.* [37] propose the best ways to undertake this. Given such a domain ontology and the six SE requirements from the previous section, we sketch features of the analysis and design phases for an ontology-based MAS methodology.

The six SE requirements are collated here for convenience:

1. Ontology mappings to integrate problem solvers of individual agent.
2. Verification of PSM knowledge requirements against ontologies at design time.

3. Knowledge extensibility is required at the agent level at least, this usually creating inconsistencies.
4. Structured and understood knowledge representation to resolve inconsistencies.
5. Iteration between lower level design (of the problem solver design) and goal analysis.
6. A consideration of the total PSMs of all agents.

There is inter-play between the role of reuse and other roles of ontologies in an MAS. Various reuse roles cannot be smoothly accommodated (e.g. interoperability at run-time) without careful consideration of run-time temporal requirements. For example, an ontology's role in reasoning at run-time is based on fulfilling PSM knowledge requirements at design time. This requires scoping domain analysis for each individual agent at design time (towards requirement 2).

Requirement 2 recognizes that the key to ontology-based design of an MAS is the appropriate allocation of a PSM to individual agents in order to match system requirements. Towards this, we note that goal analysis is the usual way to express requirements e.g. see [1, 3], and we suggest associating PSMs (using PSM libraries) and system goals in the early stages of an MAS design. The rest of the system can then be developed with appropriate ontological mappings (Fig. 2).

The collection of all PSMs for local goals should also be verified for completeness against stated system goals. These goals should also be checked against cooperation potential. (A form of distributed goal interaction evaluation could be done using approaches such as [38].) Most current methodologies view the decision of problem-solving mechanisms as a low level design step. In our current view, paralleling KBS development, ontology-based design and development requires elevating this to an early design phase and making it central to a later decision on the communication and interface requirement of each agent (rather than the other way around as in many other methodologies e.g. [1, 3]). This elevation of reasoning and iterative verification with goal analysis is one way to satisfy Requirements 5 and 6 (see Fig. 2).

Chosen problem solving capabilities for different agents in a given MAS do not necessarily have required degree of domain dependence. Hence, for a PSM chosen for some agents, the ontology required may need to be adapted. For this, the domain ontology is again the most convenient reference point. Ontology mapping (between portions of domain ontology and the local agent's knowledge) is required to ensure that all PSMs have their knowledge requirement available to their reasoning format (adaptors of Fensel [39] may be useful here). Agents need to communicate their results and instigate cooperation using a common language. For this purpose, we recommend a global communication ontology (as in [40]), rather than many-to-many individual mappings between agents. Such a communication ontology is most conveniently based on the domain ontology available and depends on the individual ontology of each agent. In some cases, an ontology mapping may be required between PSM ontologies and the communication ontology. The same adaptation between the reasoning and domain ontology can be used to map the result of reasoning back to a common communication ontology (based on the domain ontology). Our work so far is geared towards 'extendable closed' systems. In the case of 'open systems', introducing new agents may require at *runtime* extending the communication ontology or some local ontologies to allow cooperation with new agents. This is currently beyond

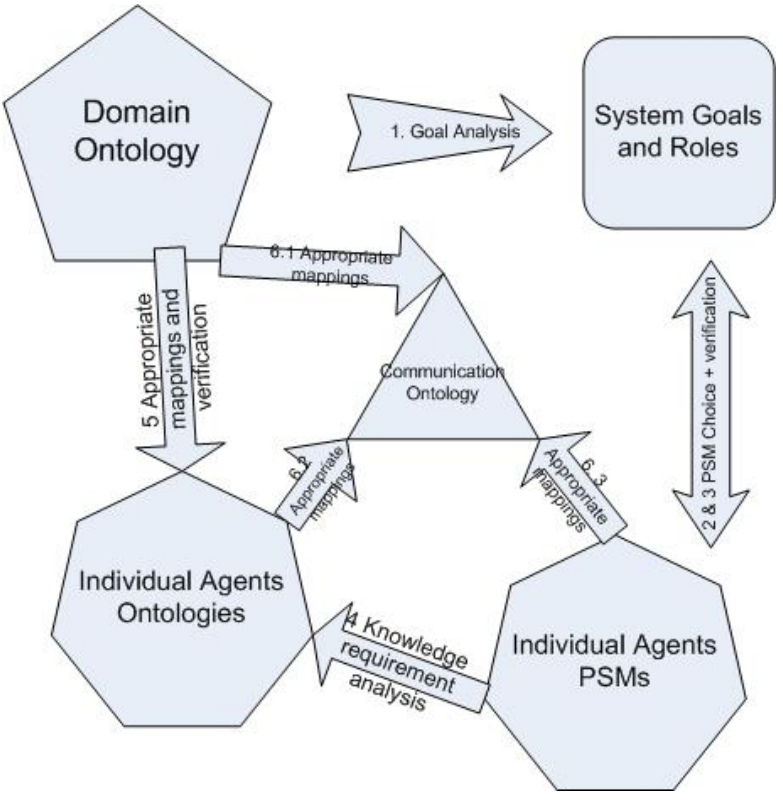

Fig. 3. 1. Ontology-based MAS development: Domain Ontology produces Goal Analysis 2. Goal analysis produces a collection of PSMs (using a PSM bank) 3. Knowledge requirement analysis (4). can then be used to delineate local ontologies that can be verified against the domain ontology (step 5). Finally, in step 6 the communication ontology (language) can then be derived using appropriate mappings.

our current scope. However, we note that we never assume that local ontologies for agents are complete from the perspective of the agent (see Section 2). This is a considerable step in the right direction towards implementing completely 'open systems'.

Towards requirements 3 and 4, hierarchical ontologies are one way to have flexible domain ontology refinement for agents according to their PSMs, and to accommodate differences in strength of the PSM of agents. A common hierarchical domain ontology can be used as a starting point for verification during development and for multiple access at multiple abstraction levels depending on the individual knowledge requirement of each agent PSM. For this purpose, Multiple Hierarchical Restricted Domain (MHRD) ontologies, employed by many authors (e.g. [41]), are well understood and expressive for most domains. MHRD models are sets of inter-related concepts that are defined through a set of attributes, so the presence of axioms between these attributes is not considered. There can be *part-of* and taxonomic relations among the concepts so that attribute (multiple) inheritance is permitted.

Fig. 3 provides a methodological sketch accommodating the observations of this section. The MAS development process starts with a domain ontology. This is used to identify goals and roles. These are used to index an appropriate set of PSMs from a bank of PSMs (similar to Fig. 2). Appropriate individual ontologies for each PSM are extracted from the initial ontology. These ontologies are used for reasoning by individual problem solvers and may be used to represent results communicated by the individual problem solver. They are next verified against the knowledge requirement of chosen PSMs. The collection of these ontologies is then used to develop a common communication ontology. Appropriate mappings may be needed between individual local ontologies and the communication ontology, to facilitate communicating results between individual agents. Verification between problem solvers and the communication ontology is undertaken, which may result in further localized ontology mappings.

5 Discussion, Conclusion and Future Work

In this paper, we have evaluated the goal of long term reuse of software engineering knowledge and effort involved in developing MASs. This reuse may take the form of extending functionality of an existing system, reusing components of an existing system in an entirely different context or creating a new system using the design (in whole or part) of an existing system. We have argued that an ontology-founded MAS methodology can produce reusable MAS components and designs. This issue of reuse is often overlooked in the MAS software engineering community. Moreover, we have argued that an ontology-based MAS methodology can truly become a domain-independent methodology by combining domain-dependent concerns of existing methodologies.

The current use of ontologies in MAS methodologies is limited to the early analysis phases and, in other cases, to express the communication languages for agents within the system. Current usage ignores the impact of using ontologies for the late design phase when components of the system begin to emerge. Taking into account this impact, we have highlighted several software engineering requirements for ontology-based multi-agent systems development. We have drawn from lessons of the knowledge-based systems (KBS) and engineering communities to use the separation of problem solving methods and ontology as a basis for reuse. As a conclusion of our analysis, we have sketched an MAS ontology-based methodology that assumes that an initial domain ontology is available. This methodology guides the allocation of individual ontologies and problem solving capabilities to individual agents in the system. To complete our sketched methodology, domain-dependence of some of its steps described in Section 4 should be recognized. An example of where this may occur is during the step producing goal analysis from the initial domain ontology, in order to index individual agents PSM. In other words, we acknowledge that it is not wise to assume that all domain dependencies are bundled in the PSM bank. Cordi *et al.* [37] explain the best way to undertake such domain-dependent model conversions. Our sketched methodology also requires the development of appropriate interfaces to PSM and ontology banks. Such banks already exist e.g. [42].

As for the later phases of our sketched methodology, there are a number of existing agent-oriented methods with differing concerns and assumptions that can be combined to produce a largely domain-independent unified approach. The result would be

a comprehensive framework that addresses the ontology concerns elucidated here and combines all domain-dependent techniques. This would produce the equivalent of PSM banks, but for the MAS software development process itself. We are currently examining different ways to unify all domain-dependent concerns of existing methodologies and interleave the domain-independent ontological SE guidelines as outlined in this paper. Metamodelling based method engineering as we outlined in [43] is particularly promising as are the very recently published foundational ontological ideas of [44].

Acknowledgment

The work is supported by the Australian Research Council under Discovery grant number: DP0451213.

References

1. Giunchiglia, F., Mylopoulos, J., Perini, A.: The Tropos Software Development Methodology: Processes, Models and Diagrams, in *Agent-Oriented Software Engineering III: Third International Workshop, AOSE 2002*, Giunchiglia, F., Odell, J. and Weiß, G., Editors. Springer (2003). 162-173.
2. Padgham, L., Lambrix, P.: Agent Capabilities: Extending BDI Theory, in *17th National Conference on Artificial Intelligence (AAAI-2000)*. Austin, Texas USA: MIT Press (2000)
3. Wooldridge, M., Jennings, N.R., Kinny, D.: The Gaia Methodology for Agent-Oriented Analysis and Design, in *Autonomous Agents and Multi-Agent Systems*. The Netherlands: Kluwer Academic Publishers (2000)
4. Tran, Q.N., Low, G.: Comparison of Methodologies, in *Agent-Oriented Methodologies*, Henderson-Sellers, B. and Giorgini, P., Editors. Idea Group Publishing: Hershey (2005) 341-367.
5. Russell, S., Norvig, P.: *Artificial Intelligence, A modern Approach, the intelligent agent book*, Prentice Hall (2003)
6. Wooldridge, M.: *Reasoning About Rational Agents*. MIT Press (2000)
7. Gruber, T.R.: A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, **5** (1993) 199-220
8. Beydoun, G., Breis, J.T.F., Béjar, R., Hoffmann, A.: Statistical Monitoring of Ontology Integration for Corporate Memory, in *Pacific Rim Knowledge Acquisition Conference (PKAW20002)*. Japan (2002)
9. Farquhar, A., Fikes, R., Rice, J.: The Ontolingua Server: a tool for collaborative ontology construction. *International Journal of Human-Computer Studies*, **46** (1997) 707-727.
10. Mukherjee, R., Dutta, P.S., Sen, S.: Analysis of domain specific ontologies for agent-oriented information retrieval, in *AAAI-2000 Workshop on Agent-Oriented Information Systems*. Austin, Texas (2000)
11. Uschold, M., Grueninger, M.: Ontologies: Principles, Methods and Application. *Knowledge Engineering Review*, **11**(2) (1996) 93-195
12. Davies, J., Fensel, D., Harmelen, F.V., eds.: *Towards The Semantic Web: Ontology-driven Knowledge Management*. Wiley: London (2003)
13. Fensel, D.: The tower-of-adaptor method for developing and reusing problem-solving methods, in *European Knowledge Acquisition Workshop*. Spain: Springer-Verlag (1997)

14. Chandrasekaran, B., Johnson, T., Smith, J.: Task Structure Analysis for Knowledge Modelling. *Communications of ACM*, **35**(9) (1992) 124-137
15. Shreiber, G., Akkermans, H., Anjewierden, A., Hoog, R., Shadbolt, N., de Velde, W.V., Wielinga, B.: *Knowledge Engineering And Management: The CommonKADS Methodology*. London: The MIT Press (2001)
16. Wielinga, B., Schreiber, G., Breuker, J.: KADS: a modelling approach to knowledge engineering. *Knowledge Acquisition*, **4** (1992) 5-53.
17. Benjamins, R., Plaza, E., Motta, E., Fensel, D., Studer, R., Wielinga, B., Schreiber, G., Zdrahal, Z.: IBROW3 - An Intelligent Brokering Service for Knowledge-Component Reuse on the World Wide Web, in *Banff Knowledge Acquisition Workshop (KAW98)*. Canada. (1998)
18. Cairo, O.: The KAMET Methodology: Content, Usage and Knowledge Modeling, in *11th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW98)*. Canada: SRDG publications (1998)
19. Benjamins, R.: Problem solving methods for diagnosis and their role in knowledge acquisition. *International Journal of Expert Systems: Research and Applications*, **2**(8) (1995) 93-120.
20. Newell, A.: The knowledge level. *Artificial Intelligence*, **18** (1982) 87-127
21. Chandrasekaran, B.: Generic tasks in knowledge-based reasoning: High level building blocks for expert system design. *IEEE Expert*, **3**(1) (1986) 23-30
22. Chandrasekaran, B.: What kind of information processing is intelligence? A perspective on AI Paradigms, and a Proposal., in *Foundations of AI: A Sourcebook.*, Partridge, D., and Wilks, Y., Editors. Cambridge University Press (1988)
23. Puppe, F.: *Systematic Introduction to Expert Systems: Knowledge Representation and Problem-Solving Methods*. Berlin: Springer-Verlag (1993)
24. Motta, E.: Parametric design problem solving, in *10th Banff Knowledge Acquisition for Knowledge Based System Workshop*. Canada)1006_
25. Cairo, O., Alvarez, J.C.: The KAMET II Approach for Knowledge-Based System Construction, in *8th International Conference on Knowledge-Based Intelligent Information and Engineering Systems (KES 2004)*. New Zealand: Springer (2004).
26. Schreiber, G., Wielinga, B., Akkermans, J., de Velde, W.V., Hoog, R.: CommonKADS: A comprehensive methodology for KBS. *IEEE Expert*, **9**(6) (1994) 28-37
27. Bernon, C., Gleizes, M.-P., Peyruqueou, S., Picard, G.: ADELFE, a Methodology for Adaptive Multi-Agent Systems Engineering, in *Engineering Societies in the Agents World*. Spain (2002)
28. Cossentino, M., Potts, C.: A CASE tool supported methodology for the design of multi-agent systems, in *International Conference on Software Engineering Research and Practice (SERP'02)*. Las Vegas (NV), USA (2002)
29. Girardi, R., Serra, I.: Using ontologies for the specification of domain-specific languages in multi-agent domain engineering, in *CAiSE Workshops (2) 2004*. (2004)
30. Dileo, J., Jacobs, T., Deloach, S.: Integrating Ontologies into Multi-Agent Systems Engineering, in *4th International Bi-Conference Workshop on Agent Oriented Information Systems (AOIS2002)*. Italy (2002)
31. Girardi, R., de Faria, C.G., Balby, L.: Ontology-based Domain Modeling of Multi-Agent Systems, in *OOPLSA Workshop* (2004)
32. Brandao, A.A.F., de Silva, V.T., de Lucena, C.J.P.: Ontologies as Specification for the Verification of Multi-Agent Systems Design, in *Object Oriented Programmings, Systems, Languages and Applications Workshop (2004)*. California (2004)

33. Esteva, M.: Electronic Institutions: From Specification To Development, in *Artificial Intelligence Research Insitute*. UAB - Universitat Autonòma de Barcelona: Barcelona (2003)
34. Padgham, L., Winikoff, M.: Prometheus: A Methodology for Developing Intelligent Agents, in *Agent-Oriented Software Engineering III: Third International Workshop (AOSE 2002, Italy)*, Giunchiglia, F., Odell, J., and Weib, G., Editors. Springer (2002) 174-185
35. Beydoun, G., Hoffmann, A., Breis, J.T.F., Martinez-Béjar, R., Valencia-Garcia, R., Aurum, A.: Cooperative Modeling Evaluated. *International Journal of Cooperative Information Systems, World Scientific*, **14**(1) (2005) 45-71
36. DARPA: Ontology Repository. http://www.daml.org/ontologies/ (2000)
37. Cordi, V., Mascardi, V., Martelli, M., Sterling, L.: Developing an Ontology for the Retrieval of XML Documents: A Comparative Evaluation of Existing Methodologies, in *AOIS2004 @CaiSE04*. (2004)
38. van Lamsweerde, A., Darimont, R., Letier, E.: Managing Conflict in Goal-Driven Requirements Engineering. *IEEE Transaction on Software Engineering*, **24**(11) (1998)
39. Fensel, D.: Using Ontologies for Defining Tasks, Problem-Solving Methods and Their Mappings, in *European Knowledge Acquisition Workshop*. Spain: Springer-Verlag (1997).
40. Esteva, M., de Cruz, D., Sierra, C.: ISLANDER: an electronic institutions editor, in *International Conference on Autonomous Agents & Multiagent Systems (AAMAS02)*. Italy: ACM (2002)
41. Eschenbach, C., Heydrich, W.: Classical mereology and restricted domains. *International Journal of Human-Computer Studies*, **43** (1995)723-740
42. Fensel, D., Benjamins, V.R., Motta, E., Wielinga, B.: UPML: A framework for knowledge system reuse, in *Sixteenth International Joint Conference on Artificial Intelligence (IJCAI99)*. Sweden: Morgan Kaufmann Publishers (1999)
43. Beydoun, G., Gonzales-Perez, C., Low, G., Henderson-Sellers, B.: Synthesis of a Generic MAS Metamodel, in *International Conference on Software Engineering (ICSE05) Workshops (SELMAS2005)*. ACM Digital Library (2005)
44. Guizzardi, G., Wagner, G.: On the ontological foundations of agent concepts, in *Agent-Oriented Information Systems II*, Bresciani, P., Giorgini, P., Henderson-Sellers, B., Low, G. and Winikoff, M., Editors. Springer (2005) 113-128

An Ontology-Driven Technique for the Architectural and Detailed Design of Multi-agent Frameworks

Rosario Girardi and Alisson Neres Lindoso

Federal University of Maranhão, Portugueses Av., Campus do Bacanga,
65.080-480, São Luiz-MA, Brazil
rgirardi@deinf.ufma.br, alissonlindoso@uol.com.br

Abstract. Multi-agent Domain Engineering is a process for the construction of domain-specific agent-oriented reusable software artifacts, like domain models, representing the requirements of a family of multi-agent systems and frameworks, implementing an agent-oriented solution to those requirements. This work describes DDEMAS, an ontology-driven technique for the architectural and detailed design of multi-agent frameworks providing a solution to the requirements of a family of multi-agent software systems specified in a domain model. DDEMAS is part of MADEM, a methodology for domain analysis, design and implementation of a family of multi-agent systems in a domain. Domain models and multi-agent frameworks are part of a knowledge base constructed through the instantiation of ONTOMADEM, an ontology that represents the knowledge of MADEM. Some examples from a case study on the application of DDEMAS on the construction of a multi-agent framework model for the development of usage mining-based Web recommender systems are also described.

1 Introduction

Considerable advances on the systematization of the agent-oriented development paradigm have been achieved and several techniques, methodologies and software development environments are already available for the development of multi-agent applications [1-8]. Some methodologies promote the reuse of software patterns [3], although little work has been done on the development of techniques and methodologies for the construction of high-level reusable software abstractions in this development paradigm.

Domain Engineering and Application Engineering [9-10] are two complementary software processes. Domain Engineering, also known as Development *for* Reuse, is a process for creating software abstractions reusable on the development of a family of software systems in a domain, and Application Engineering or Development *with* Reuse, for constructing a specific application using reusable software abstractions available in the target domain(s). A family of systems is defined as a set of existing software systems sharing some commonalities but also particular features [9].

The process for Domain Engineering is composed of the phases of analysis, design and implementation of a domain. Domain analysis activities identify reuse opportunities and determine the common and variable requirements of a family of applications. The product of this phase is a domain model. Domain design activities

M. Kolp et al. (Eds.): AOIS 2005, LNAI 3529, pp. 124 – 139, 2006.

look for a documented solution to the problem specified in a domain model. The product of this phase is composed of one or more frameworks and, possibly, a collection of design patterns, documenting good solutions in that domain. Reusable components integrating the framework are constructed during the phase of domain implementation. This is the compositional approach of Domain Engineering. In a generative approach, Domain Engineering produces Domain Specific Languages (DSLs) and application generators to construct a family of applications in a domain. Knowledge of the domain and design patterns are encoded in DSLs [9, 11].

Ontologies [12] are knowledge representation structures particularly useful for the specification of high-level reusable software abstractions, providing an unambiguous terminology that can be shared by all involved in a development process. Ontologies can also be as generic as needed allowing its reuse and easy extension.

A collection of ontology-based reusable software abstractions is being developed in the context of a Multi-Agent Domain and Application Engineering research project [11, 13, 14]. The multi-agent paradigm has been adopted because of its effectiveness to approach software complexity.

This work describes the DDEMAS technique for the architectural and detailed domain design of multi-agent systems. The technique is part of MADEM ("Multi-Agent Domain Engineering Methodology"), an ontology-based methodology that provides support for all the phases of the Multi-agent Domain Engineering process. MADEM also integrates GRAMO [14], a technique for domain analysis of multi-agent systems. Previous work on the DDEMAS technique has been already published [15]. MADEM is supported by ONTOMADEM, an ontology-driven tool in which MADEM products are represented as instances of an ontology that conceptualizes the methodology.

The paper is organized as follows. Section 2 summarizes the modelling phases and respective tasks of MADEM. Section 3 details the architectural and detailed design phases of DDEMAS. Section 4 discusses related work on this research topic. Section 5 concludes the paper with some remarks on further work being conducted.

2 An Overview of the MADEM Methodology

Modelling concepts, tasks and products of MADEM are based on techniques for Domain Engineering [9], development of multi-agent systems [1-7] and software pattern specification and reuse [16-17].

For the specification of the problem domain to be solved, MADEM focusses on modelling goals, roles, activities and interactions of entities of an organization. Entities have knowledge and use it to exhibit autonomous behaviour.

An organization is composed of entities with general and specific goals that establish what the organization intends to reach. The achievement of specific goals permits the attainment of the general goal of the organization. For instance, an information system can have the general goal "satisfying the information needs of an organization" and the specific goals of "satisfying dynamic or long term information needs". Specific goals are reached through the performance of responsibilities that entities have by playing roles with a certain degree of autonomy.

Responsibilities are exercised through the execution of activities. The set of activities associated with a responsibility are a functional decomposition of it.

Roles have skills on one or a set of techniques that support the execution of responsibilities and activities in an effective way. Pre-conditions and post-conditions may need to be satisfied for/after the execution of an activity. Knowledge can be consumed and produced through the execution of an activity. For instance, an entity can play the role of "information retriever" with the responsibility of executing activities to satisfy the dynamic information needs of an organization. Another entity can play the role of "information filter" with the responsibility of executing activities to satisfy the long-term information needs of the organization. Skills can be, for instance, the rules of the organization that entities know to access and structure its information sources.

Sometimes, entities have to communicate with other internal or external entities to cooperate in the execution of an activity. For instance, the entity playing the role of "information filter" may need to interact with a user (external entity) to observe his/her behaviour in order to infer his/her profile of information interests.

For the specification of a design solution, responsibilities of roles are assigned to agents structured and organized into a particular multi-agent architectural solution according to non-functional requirements.

Fig. 1 illustrates the phases of the MADEM methodology in the context of a Multi-agent Domain Engineering process, and Table 1 summarizes their modelling phases, respective tasks and modelling products. The framed concepts of Fig. 1 illustrate the phases and reusable products generated through the application of the DDEMAS technique of MADEM.

Domain Analysis supported by the GRAMO technique approaches the specification of current and future requirements of a family of applications in a domain by considering domain knowledge and development experiences extracted from domain specialists and applications already developed in the domain. Existing analysis patterns can also be reused in this modelling task.

Domain Analysis is performed through the following modelling tasks: *Concept Modelling*, *Goal Modelling*, *Role Modelling*, *Variability Modelling* and *Role Interaction Modelling* (Table 1). In this phase, a *Domain Model* is obtained through the composition of:

- a *Concept Model*, representing a first draft of concepts in the problem domain and relationships between them;
- a *Goal Model*, specifying the general and specific goals of the system family along with the responsibilities required to achieve the goals;
- a *Role Model*, specifying the roles in charge of responsibilities; activities that need to be executed to exercise a responsibility; skills required for exercising a responsibility, pre- and post-conditions that must be satisfied before and after the execution of an activity; and, finally, the knowledge required/produced for/from the execution of the activities.
- a set of *Role Interaction Models* specifying the interactions between roles and external entities needed to achieve a specific goal.

The *Goal* and *Role Models* provide a static view of the organization; the set of *Interactions Models*, a dynamic one.

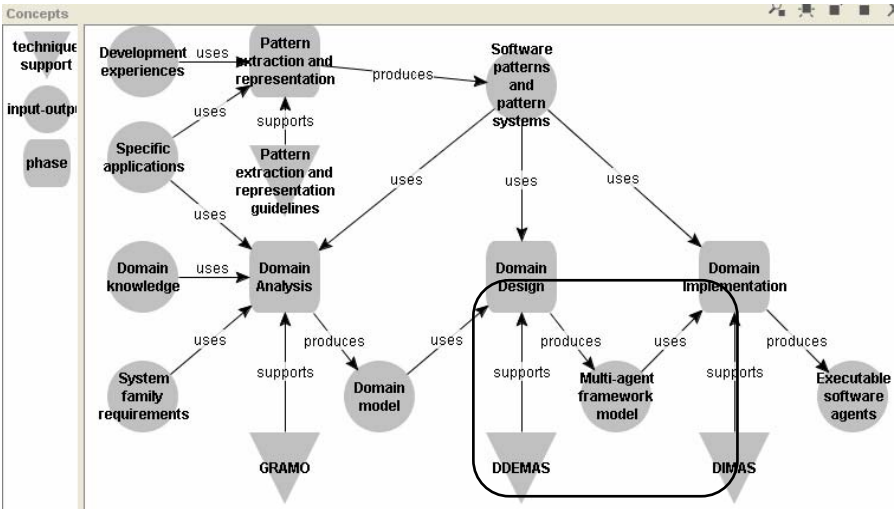

Fig. 1. MADEM methodology in the context of the Multi-agent Domain Engineering process

Table 1. Modelling phases, tasks and products of the MADEM methodology

Phases	Tasks		Products		
Domain Analysis	Modelling of domain concepts		Concept Model	Domain Model	
	Variability Modelling	Goal Modelling	Goal Model		
		Role Modelling	Role Model		
	Modelling of Role Interactions		Role Interaction Models		
Domain Design	Architectural Modelling	Agent Society Modelling	Multi-agent Society Model	Architectural Model	Model of the multi-agent framework
		Agent Interaction Modelling	Agent Interaction Model		
		Cooperation and Coordination Modelling	Coordination and Cooperation Model		
	Modelling the Knowledge of the Multi-agent Society		Model of the Multi-agent Society Knowledge		
	Agent Modelling		Agent Knowledge and Activity Models	Agent Models	
			Agent State Models		
Domain Iplementation	Mapping from design to implementation agents and behaviours		Model of agents and behaviours		Implementation Model of the Multi-agent Society
	Mapping from agent interactions to communication acts		Model of communication acts		
Pattern extraction and representation			Software Patterns and Pattern Systems		

Domain design supported by the DDEMAS technique targets the architectural and detailed design of multi-agent frameworks providing a solution to the requirements of a family of multi-agent software systems specified in a domain model. The DDEMAS technique is the main topic of this paper (described in Section 3).

Domain implementation supported by the DIMAS technique approaches the mapping of design models to agents, behaviours and communication acts [23], concepts involved in the JADE framework [19], which is the adopted implementation platform. An *Implementation Model of the Multi-agent Society* is constructed as a product of this phase of MADEM, composed of a *Model of agents and behaviours* and a *Model of communication acts.*

In the Pattern extraction and representation phase, MADEM provides guidelines for the extraction of patterns and systems of patterns from available software applications and considering successful development experiences.

3 The DDEMAS Technique

The DDEMAS technique approaches the architectural and detailed domain design of multi-agent systems guiding a set of modelling tasks for the construction of a framework model of the multi-agent society.

The technique consists of three sub-phases:

- *Architectural modelling*, for the construction of the architecture of a family of multi-agent systems.
- *Modelling of the knowledge of the multi-agent society*, for representing the meaning of concepts which agents in the society need to understand in order to communicate with each others.
- *Agent modelling*, for the construction of the internal architecture of each agent in the society.

The following sub-sections detail the tasks performed in each sub-phase of DDEMAS illustrated with examples extracted from a case study on the development of ONTOWUM, a family of multi-agent Web recommender systems based on usage mining and collaborative filtering [20].

3.1 Architectural Modelling

The purpose of the *Architectural modelling* task is to develop an *architectural model* representing an agent-oriented architectural solution to the problem specified in a domain model. This architectural model is composed of three sub-products: a *Multi-agent society model*, an *Agent Interaction model* and a *Coordination and Cooperation model* developed, respectively, through the tasks *Agent Society Modelling*, *Modelling the interactions of the multi-agent society* and *Modelling of Cooperation and Coordination*, described as follows.

Agent Society Modelling. The purpose of this subtask is to identify and represent the agents that will populate the multi-agent society and represent them in a *Multi-Agent Society Model*. The agents are identified from the roles specified in the *Role Model* and *Role Interaction Models* of the *Domain Model*. Initially, each "role" maps to one

"agent". Agent variability is derived from role variability. Responsibilities, activities, knowledge, conditions and skills required for performing the role are also mapped to the corresponding agent. Then, design rules like functional cohesion may be applied and non-functional requirements must be considered (e.g. performance). Roles in charge of similar responsibilities, roles exhibiting a high number of interactions between them or roles interacting with the same external entity are candidates for fusion into one agent. In this case, the agent will perform more than one role.

The *Multi-Agent Society Model* is represented graphically in a three level organizational chart. Agents and skills are represented in the first and third level, respectively; the other concepts of modelling in the second one. Fig. 2 shows part of the *Multi-agent Society Model* of the ONTOWUM multi-agent framework model showing the *Interface* and *User Modeller* agents.

In the Role Modelling task of the Domain Analysis phase [16], the *User Monitor*, *User Modeller*, *Acquirer*, *Miner*, *Classifier*, *Customizer* and *Interface* roles were identified with, respectively, the following responsibilities:

- *User Monitoring* for the extraction of the *user browsing information on a Web page* from *the interactions of a user with a Web page*;
- *Modelling of current user* for the creation and updating of a *model of the current user*, a formal representation of the *user browsing information on a Web page*.
- *Usage data maintenance* for the creation and update of a *repository of usage data*, from the *model of the current user*;
- *Discovery of usage patterns* for the identification and representation of groups of users with similar browsing behaviour;
- *Classification of current user* for the classification of the *model of current user* in a *model of groups of users* discovered by the *Miner* role thus determining the *groups of users* to which the current user belongs;
- *Construction of adaptation model* for the creation and update of an adaptation model specifying the adaptation rules to be applied on the interface of a current user according to the groups of users to which the current user belongs;
- *Adaptation of the Web Interface* for the creation and update of a *customized user interface* according to the *adaptation model.*

According to the rules of the *Agent Society Modelling task* explained above, roles are mapped to agents and some of them are fused into one agent. An *Interface* agent will play the roles of *User Monitor* and *User Interface* since both roles interact with the *User* external entity [16].

The Interface agent. When a *new user is connected*, a new *Interface* agent is created which performs the *User Monitoring* responsibility for the extraction of the *user browsing information on a Web page*, containing information like the URL of a visited page and the time spent on that page from *the interactions of a user with a Web page.* Among the alternative skills specified in the domain model of ONTOWUM [16] for user monitoring, the approach of Shahabi *et al.* [22] for client side data acquisition through Java applet remote-agents was chosen.

When an adaptation model is available, the *Interface* agent performs the *Adaptation of the Web Interface* responsibility for the creation and update of a *customized user interface* according to the *adaptation model.* The customization is

based on a set of recommended links, which are displayed in a frame of the Web page. As soon as there are new recommendations, the frame becomes visible showing the current recommendations to the user.

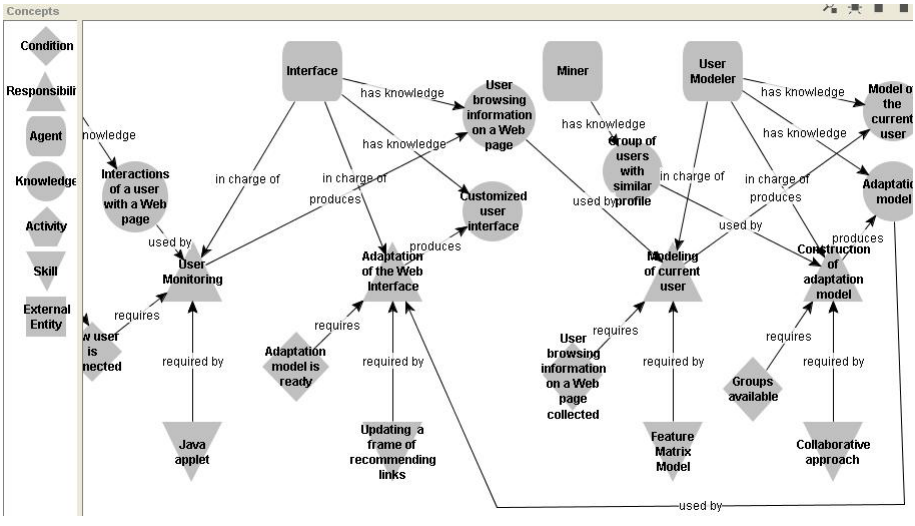

Fig. 2. Part of the *Multi-agent Society Model* of the ONTOWUM multi-agent framework model showing the *Interface* and *User Modeller* agents

The User Modeller agent. For each new user connected to the system, a *User modelling* agent is created. When the pre-condition *User browsing information on a Web page collected* is satisfied, the *User Modeller* agent performs the *Modelling of current user* responsibility which involves the creation and updating of a *model of the current user* with the *user browsing information on a Web page*, captured by the *Interface* agent. Among the alternative skills specified in the domain model of ONTOWUM [16], the Feature Matrix model [22] was used as the technique to formally represent the user models of the system where features are the visited URLs and time of page view.

Modelling the interactions of the multi-agent society. The purpose of this subtask is to identify the interactions between agents needed to accomplish their responsibilities. For that, initially, the interactions between roles in the *Role Interaction Models* of the *Domain Model* are refined according to the agents specified in the *Multi-agent Society Model*. Through an analysis of the agent responsibilities and activities along with their required and produced knowledge, the interactions between agents and between agents and external entities are identified and represented in an *Agent Interaction Model* whose graphical representation and semantics are similar to the interaction diagram of AUML [5]. Agent interactions are represented using the performatives of FIPA-ACL [23]. The events provoked by actions of external entities and perceived by agents are also represented in this diagram.

Fig. 3 shows the *Agent Interaction Model* of the ONTOWUM multi-agent framework model. When a *new user is connected*, a new *Interface* agent and a new *User Modeller* agent are created to act on behalf of this user. By performing the *User monitoring* responsibility, the *Interface* agent captures the URL and the time spent visiting a Web page and informs this browsing behaviour to the *User Modeller* agent. The *Interface* agent communicates with the *User Modeller* agent each time a user loads a new page on the same site. The *User Modeller* agent then creates a model of the current user, if it is the first page visited on the site or updates it with the new pages visited. Then, it informs the *Miner* agent about the new model of the current user. When the system is initialized, a *Miner* and an *Acquirer* agent are created.

When the current user leaves the site or closes the browser, the *User Modelling* agent is notified by the *Interface* agent and then it informs the *Acquirer* agent of the model of the current user. The *Acquirer* agent will use this model to create a repository of usage data, if this is the first user that has connected to the system, or update it, otherwise.

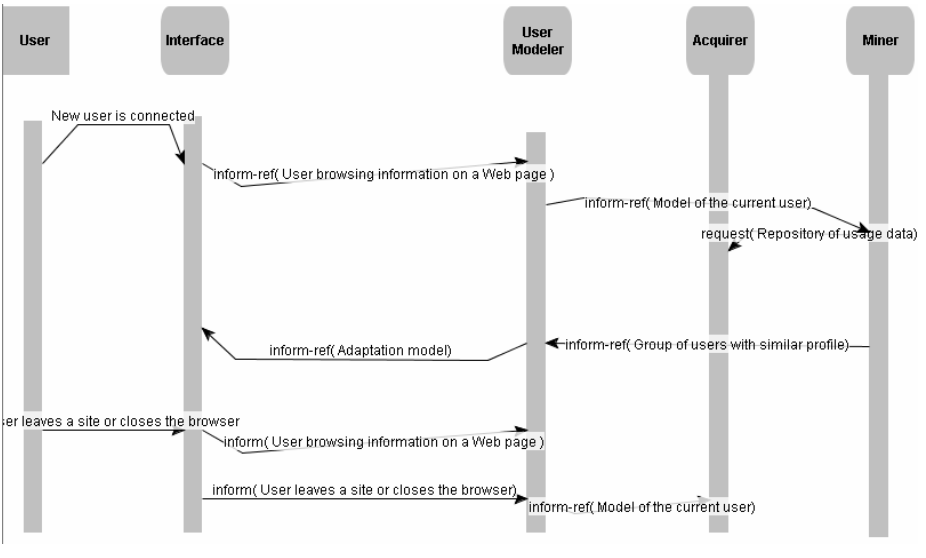

Fig. 3. The Agent Interaction Model of the ONTOWUM multi-agent framework model

Once the *Miner* agent has classified the current user in one of the groups of users previously discovered, it informs the *User Modeller* agent about this group. Then, the *User Modeller* agent will construct an adaptation model with the rules associated with that group and will inform this model to the *Interface* agent, which will perform the customization of the user interface.

The repository of usage data maintained by the *Acquirer* agent is periodically consulted by the *Miner* agent for the identification and representation of groups of users with similar browsing behaviour.

Modelling of Cooperation and Coordination. From the first drafts of the *Multi-agent Society Model* and *Agent Interaction Models*, and considering available architectural patterns [17] and/or appropriate mechanisms of cooperation and coordination [24-25], the agent society is organized in a *Cooperation and Coordination Model*.

For the identification of an architectural pattern, the descriptions of general and specific goals in a *Goal Model* of a *Domain Model* produced in the domain analysis phase of the MADEM methodology are matched with the *problem* attribute in a pattern description or the *description* attribute in a pattern system. A *Goal model* specifies the problems and sub-problems that an MAS is intended to solve (Table 1).

Obviously, selected patterns must have a context description related to the architectural design of multi-agent systems. If a total or partial match is obtained, the solution described in the pattern is considered for the execution of the other tasks of architectural design.

Through this reorganization, two or more agents can fuse or one agent can be divided in two or more agents. These changes are represented in a new *Agent Society Model* and new *Agent Interaction Models*.

Fig. 4 shows the *Coordination and Cooperation Model* of the ONTOWUM architectural model, where a two-layer architecture and the solution suggested by the WUMA_MAS architectural pattern [17] is adopted to organize the agents that populate the framework ONTOMUW. The model follows the architectural design of a multi-agent layer pattern [17]. The architectural organization of multi-agent systems in layers contributes to the understanding, reuse and maintenance of the systems. The alteration of a layer affects at most two other layers, because a layer communicates, at least, with a layer and, at most, with two. The reuse of layers is promoted because similar responsibilities are grouped in a layer.

The solution (Fig. 4) involves the definition of a criterion for the division of the responsibilities between agents distributed in layers, according to the *Multi-agent Layer* pattern [17]: a *Discovery of user navigational patterns* layer, responsible for discovering groups of users with similar navigational behaviour, which provides services to a *User information processing* layer in order to offer customized services to the end user.

A group of agents is associated to each layer. Each layer is structured as follows.

User information processing layer. In this layer, one interface agent and one user modelling agent are associated with each user. These agents are structured according to the *WUMA - Interface* and *WUMA - UserM* patterns [17], respectively, supporting the following responsibilities:

- Constructing and displaying a personalized interface;
- Capturing implicitly the user profile.
- Creating and updating a user model representing the user profile.

Discovery of user navigational patterns layer. This layer is composed of one acquirer and one miner agent structured according to the *WUMA - Acquirer* and *WUMA - Miner* patterns, respectively, supporting the following responsibilities:

- Recording and updating a usage data repository.
- Discovering groups of users with similar navigation behaviour and identifying to which group a current user belongs.

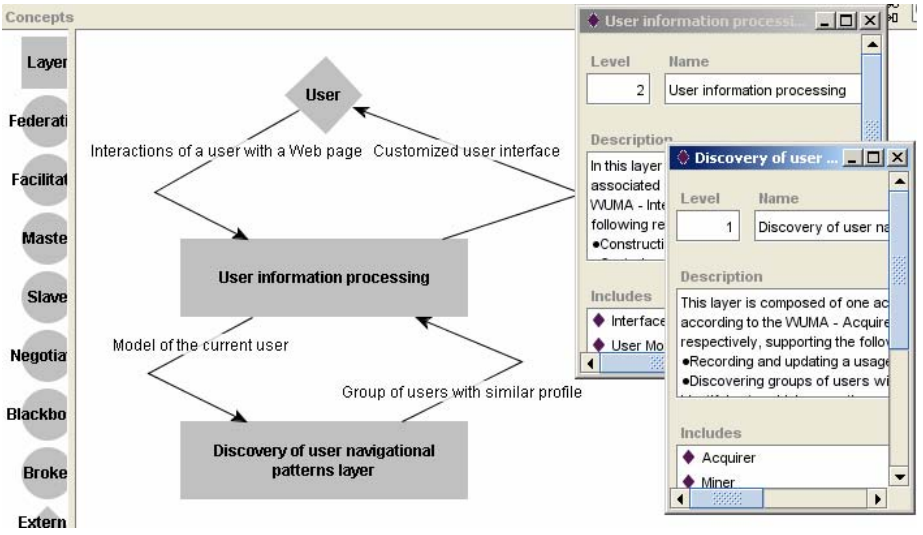

Fig. 4. Coordination and Cooperation Model of the ONTOWUM architectural model based on a multi-agent layer pattern

Modelling the Knowledge of the Multi-agent Society. The purpose of this task is to represent the meaning of concepts that agents of the society need to understand in order to communicate with each other. This is done through the construction of a model of the multi-agent society knowledge, represented in a semantic network. For that, the techniques specified as skills for the execution of agent activities in the *Agent Society Model* (e.g. Fig. 2) are analysed and a basic vocabulary is defined for each skill. When an *Agent Society Model* is constructed in ONTOMADEM, the skills represented in the model are instantiated with a detail description and related literature on the corresponding techniques. The analysis of the techniques to be represented in the *Knowledge Model of the Multi-agent Society* is then based on these descriptions and related literature.

Each term in the vocabulary is represented as a node in the semantic network. The relationships between the different concepts in the vocabulary are also defined and represented as links in the semantic network. Note that variable skills (alternative or optional) produce alternative semantic networks.

Fig. 5 shows an example of a *Multi-agent Society Knowledge Model*, which is part of the framework ONTOMUW [20] and specifies the semantics of concepts involved in a Web usage mining process. For the discovery of groups of similar users over unlabelled usage data, a clustering technique is used [22]. For that, it is necessary to match each individual user model with each other in order to construct groups with the most similar users. For the construction of the individual user models to be further matched, the Feature Matrix model [22] is used. Note that these concepts are captured through an analysis of the Feature Matrix model specified as a resource of the User Modeller and Miner agents of the *Agent Society Model* of the domain model of ONTOWUM [16].

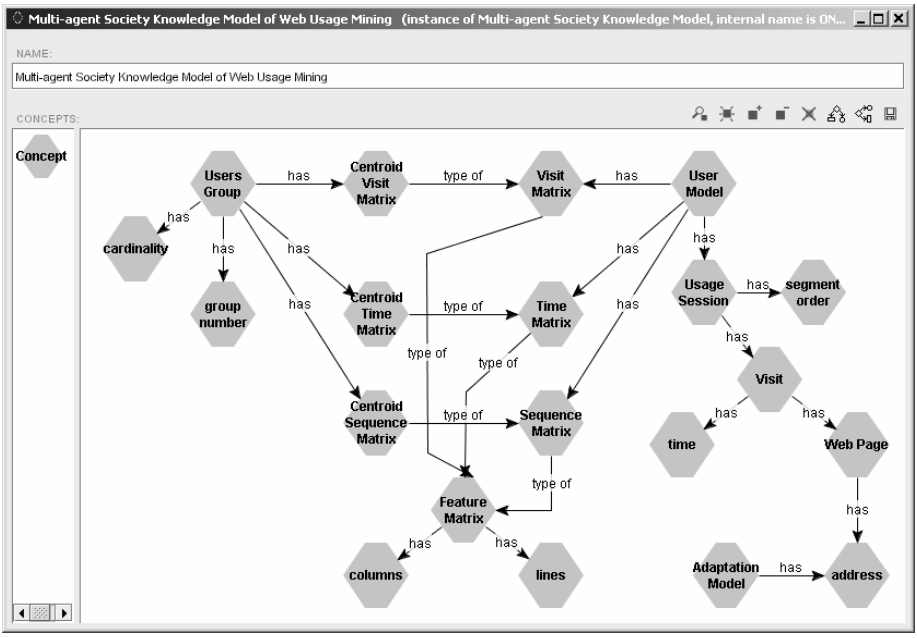

Fig. 5. Multi-agent Society Knowledge Model of the ONTOWUM framework model

3.2 Agent Modelling

The purpose of this sub-phase is to perform the detailed design of each agent in the framework, resulting in a set of agent models, each composed of an *Agent State Model* and a set of *Agent Knowledge and Activity Models*.

For each agent represented in the *Multi-agent Society Model*, an *Agent Model* is constructed. First, design patterns describing solutions for the detailed design of the agent are identified [16-17, 26-27], in a similar way as illustrated in Fig. 4 for architectural design. When patterns that apply to the design of the internal architecture of an agent are identified, the problem and forces in the selected patterns are then matched with the description of the responsibilities of the agent as specified in the *Multi-agent Society Model*. After the selection of a pattern, the agents are structured according to the solution proposed by the pattern.

If there are no reusable design patterns available for the design of an agent, a specific solution must be constructed. In this case, the type of agent (reactive or deliberative) is selected, establishing the mechanisms for mapping perceptions to agent actions [28] by considering non functional requirements (e.g. performance).

Next, from the specifications in the *Multi-agent Society Model*, *Agent Interaction Model* and *Cooperation and Coordination Model*, the possible states and state transitions of this agent are identified and represented in an *Agent State Model*. The graphical notation of the *Agent State Model* is similar to the corresponding diagram of AUML [5].

Then, for each responsibility to be exercised by the agent specified in the *Multi-agent Society Model* and for a particular set of techniques selected to perform it, a *Knowledge and Activity Model* is constructed based initially on the activities,

conditions and knowledge of that agent represented in the *Multi-agent Society Model*, refined according to the particular techniques being applied.

Fig. 6 shows the example of the Model of the *Miner Agent* of the ONTOWUM framework model. The WUMA-Miner detailed design pattern [17] has been identified and has been applied in the design of the agent.

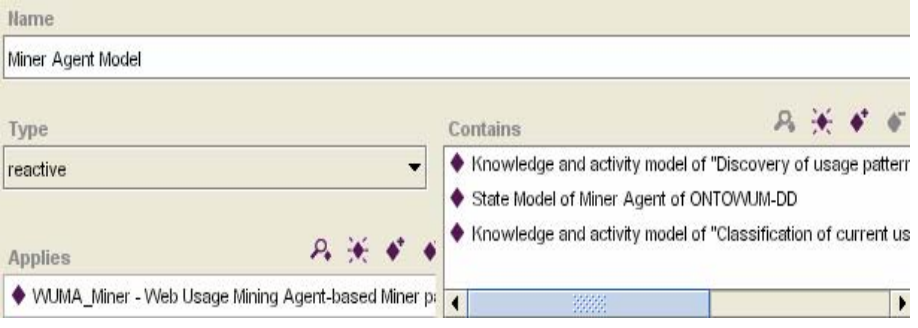

Fig. 6. Model of the Miner Agent of the ONTOWUM multi-agent framework

A reactive structure has been chosen according to the pattern guidelines and an *Agent Model* has been constructed composed of an *Agent State Model* and two *Agent Knowledge and Activity Models*, corresponding to the *Discovery of usage patterns* and *Classification of current user* responsibilities.

Fig. 7 shows the State chart diagram representing the internal behaviour of the *Miner agent*. After the system is initiated, the agent is created. As soon as the agent is created it enters the *Waiting schedule expiration* state. Since there is probably no usage data to be mined when the system is first started, the agent waits for a certain period of time in order to start the execution of the mining process.

The mining process executes periodically according to a schedule defined by the developer. After this schedule has expired, the agent enters the *Creating group models* state. Once in this state, the agent performs the extraction of user models of past users from the usage file and creates the groups, through a dynamic clustering algorithm. Once the groups are available, the agent waits for a classification requests from the user modelling agents (*Waiting user modelling agent request* state*).*

Once a request arrives from a user modelling agent, the miner agent enters the *Classifying current user* state and performs the real time classification of the current user through dynamic clustering. Note that the schedule for the mining process can expire at any time; whenever this happens the agent reinitiates the mining process.

Fig. 8 shows the Agent Knowledge and Activity Model of the Miner Agent corresponding to the responsibility Discovery of Usage patterns. Once a determined schedule has expired, the Miner Agent requires the Repository of Usage Data from the Acquirer Agent, extracts from it the models of users and represents them using the Feature Matrix Modelling technique [22]. Clustering techniques are then applied for mining user models and group them in models of groups of user with similar navigational behaviour.

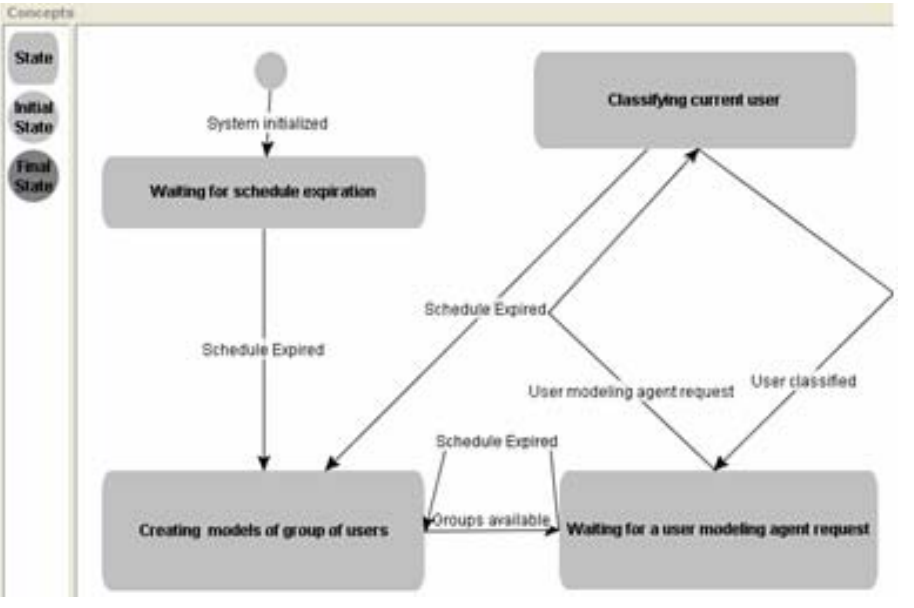

Fig. 7. Agent State Model of the Miner Agent of the ONTOWUM framework

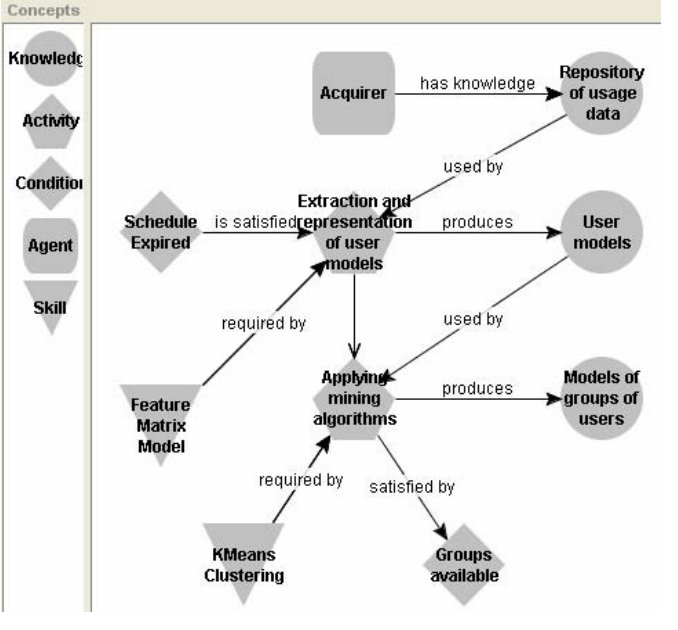

Fig. 8. Agent Knowledge and Activity Model of the Miner Agent corresponding to the responsibility Discovery of Usage patterns

4 Related Work

Several techniques for Domain Engineering [9-10] and development of multi-agent systems [3-7, 29] were analysed and have influenced in different aspects the definition of the DDEMAS technique.

Two main features distinguish DDEMAS from other existing approaches. First, it provides support for the construction of reusable agent-oriented software artifacts, and second, it is a knowledge-based technique where models of agents and frameworks are represented as instances of the ONTOMADEM ontology. Thus, concepts are semantically related allowing effective searches and inferences thus facilitating the understanding and reuse of the models during the development of specific applications in a domain.

On the other hand, some prototypes of knowledge-based tools and environments [30] have been already developed to increase the productivity of the software development process, the reusability of generated products and the effectiveness of project management. One main characteristic distinguishing MADEM from these approaches is its reuse support for agent-oriented software development.

5 Concluding Remarks

This work has introduced DDEMAS, an ontology-based technique for Domain Design in Multi-agent Domain Engineering. The technique approaches the construction of frameworks to be reused on the development of multi-agent software applications.

Frameworks are embedded in a knowledge base and created through the instantiation of the hierarchy of classes of ONTOMADEM, an ontology that represents the knowledge of MADEM. This is a methodology that integrates DDEMAS to GRAMO and DIMAS, techniques for Domain Analysis and Implementation of Multi-agent Domain Engineering.

Using MADEM, a case study has been developed where a domain model and a multi-agent framework of a family of multi-agent Web recommender applications based on usage mining have been constructed [16, 20]. From this experience, a system of architectural and design patterns for that problem-solving area has been extracted [17] and classified in the ONTOPATTERN ontology [31]. ONTO-PATTERN collects general and specific problem solving patterns for agent-oriented software development. These models and patterns can be reused in the development of recommendation systems for the legal domain according to the techniques for Multi-Agent Application Engineering we are currently developing [32].

Acknowledgments

This work has been supported by CNPq, an institution of the Brazilian Government for scientific and technologic development.

References

1. Bresciani, P. *et al.*: TROPOS: An Agent-Oriented Software Development Methodology. Journal of Autonomous Agents and Multi-Agent Systems, Kluwer Academic Publishers Vol. 8, N. 3 (2004) 203 – 236.
2. Caire, G. *et al.*: Agent-Oriented Analysis using MESSAGE/UML. In: Lecture Notes in Computer Science, Vol. 2222. Springer-Verlag, Berlin Heidelberg New York (2002) 119-135.
3. Cossentino, M. *et al.*: Patterns reuse in the PASSI methodology. In: Lecture Notes in Computer Science, Vol. 3071. Springer-Verlag, Berlin Heidelberg New York (2004) 294-310.
4. Dileo, J., Jacobs, T. and Deloach, S.: Integrating Ontologies into Multi-Agent Systems Engineering. In: Proceedings of 4th International Bi-Conference Workshop on Agent Oriented Information Systems, CEUR, Vol, 59 (2002) 15-16.
5. Odell, J., Parunak, H.V.D. and Bauer, B.: Representing Agent Interaction Protocols in UML. In: In: Lecture Notes in Computer Science, Vol. 1957. Berlin Heidelberg: Springer-Verlag (2000) 3-17.
6. Omicini, A.: SODA Societies and Infrastructures in the Analysis and Design of Agent-based Systems. In: Lecture Notes in Computer Science, Vol. 1957. Berlin Heidelberg: Springer-Verlag (2001) 185-194.
7. Wooldridge, M., Jennings, N. and Kinny, D.: The Gaia Methodology for Agent-Oriented Analysis and Design. In: International Journal of Autonomous Agents and Multi-Agent Systems, Kluwer, Vol. 3 (2000) 285-312.
8. Zambonelli, F. and Omicini, A.: Challenges and Research Directions in Agent-Oriented Software Engineering. Autonomous Agents and Multi-Agent Systems, Vol. 9, N. 3 (2004) 253 – 283.
9. Czarnecki, K., Eisenecker, U. W.: Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley Publishing Co., New York (2000).
10. Harsu, M.: A Survey of Domain Engineering. Report 31, Institute of Software Systems, Tampere University of Technology (2002).
11. Girardi, R, Serra, I.: Using Ontologies for the Specification of Domain-Specific Languages in Multi-Agent Domain Engineering. In: Grundspenkis, J. and Kirikova, M. (eds.) Proceedings of the Sixth International Bi-Conference Workshop on Agent-oriented Information Systems at CAISE'04 (2004) 295-308.
12. Chandrasekaran, B., Josephson, J. R., and Benjamins, V. R.: What are Ontologies, and why do we need them? IEEE Intelligent Systems, Vol. 14, N.1 (1999) 20-26.
13. Girardi, R., Faria, C. and Marinho, L.: Ontology-based Domain Modeling of Multi-Agent Systems. In: Cesar Gonzalez-Perez (Ed.) Proceedings of the Third International Workshop on Agent-Oriented Methodologies at International Conference on Object-Oriented Programming, Systems, Languages and Applications (2004) 51-62.
14. Girardi, R., and Faria, C.: An Ontology-Based Technique for the Specification of Domain and User Models in Multi-Agent Domain Engineering. CLEI Electronic Journal, Vol. 7, N. 1 (2004) Pap. 7.
15. Girardi, R.; Lindoso, A.: DDEMAS: A Domain Design Technique for Multi-agent Domain Engineering". In: Proceedings of ER Workshops, Lecture Notes in Computer Science, Vol. 3770. Berlin Heidelberg: Springer-Verlag (2005) 141-150.
16. Girardi, R., Oliveira, I, and Bezerra, G.: Towards a System of Patterns for the Design of Agent-based Systems. In: Proceedings of The Second Nordic Conference on Pattern Languages of Programs" (VikingPLoP 2003). Bergen, Norway (2003).

17. Girardi, R., Marinho, L. and Oliveira, I.: A System of Agent-based Patterns for User Modeling based on Usage Mining. Interacting with Computers, Elsevier, Vol. 17, N. 5 (2005) 567-591.
18. FIPA Communicative Act Library Specification, Technical Report SC00037J (2002). http://www.fipa.org/
19. Bellifemine, F., Caire, G., Poggi, A., Rimassa, G.: JADE A White Paper. Exp Vol. 3 N. 3 (2003) 6-19.
20. Marinho, L. B.: A Multi-Agent Framework for Usage Mining and User Modeling-based Web Personalization. Master dissertation, Federal University of Maranhão - UFMA – CPGEE (2005). (In Portuguese)
21. Girardi, R. and Marinho, L.: A Domain Model of Web Recommender Systems based on Usage Mining and Collaborative Filtering. Requirements Engineering Journal, Ed. Springer-Verlag. London (2006). (to appear)
22. Shahabi, C. and Banaei-Kashani, F.: Efficient and Anonymous Web Usage Mining for Web Personalization. INFORMS Journal on Computing , Vol.15, No.2 (2003) 123-147.
23. FIPA Communicative Act Library Specification, Technical Report SC00037J (2002). http://www.fipa.org/
24. Ferber, Jacques. Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence, Addison-Wesley (1999).
25. Jennings, N. R.: Coordination Techniques for Distributed Artificial Intelligence. In: O'Hare G M P and Jennings N R (eds): Foundations of Distributed Artificial Intelligence, London, Wiley (1990) 187-210.
26. Bresciani, P. *et al.*: Agent Patterns for Ambient Intelligence. In: Atzeni, P., Chu, W., Lu, H., Zhou, Z., Wang Ling, T. (eds.): Conceptual Modeling – ER 2004: 23rd International Conference on Conceptual Modeling, Shanghai, China, November 8-12, 2004. Proceedings. Lecture Notes in Computer Science, Vol. 3288. Springer-Verlag, Berlin Heidelberg New York (2004) Chapter: p. 682.
27. Lind, J.: Patterns in Agent-Oriented Software Engineering. In: Lecture Notes in Computer Science, Vol. 2585. Berlin Heidelberg: Springer-Verlag (2003) 47-58.
28. Russell, S. and Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice-Hall (1995).
29. Bernon, C. *et al.*: ADELFE: a Methodology for Adaptive Multi-Agent Systems Engineering. In: Lecture Notes in Computer Science, Vol. 2577. Springer-Verlag, Berlin Heidelberg New York (2003) 156-169.
30. Falbo, R. A., G. Guizzardi, and Duarte, K. C.: An Ontological Approach to Domain Engineering. In: Proceedings of the XIV International Conference on Software Engineering and Knowledge Engineering, ACM Press (2002) 351-358.
31. Girardi, R. and Lindoso, A.: An Ontology-based Knowledge Base for the Representation and Reuse of Software Patterns. ACM SIGSOFT Software Engineering Notes, Vol. 31, N. 1 (2006) 1-6.
32. Drumond, L., Lindoso, A. and Girardi, R.: INFONORMA: a Recommender System based on the technologies of the Semantic Web. INFOCOMP Journal, vol. 5, n. 3 (2006).

An Ontology Support for Semantic Aware Agents

Michele Tomaiuolo, Paola Turci, Federico Bergenti, and Agostino Poggi

Università degli Studi di Parma
Dipartimento di Ingegneria dell'Informazione
Viale delle Scienze, 181A – 43100 – Parma
{tomamic, turci, bergenti, poggi}@ce.unipr.it

Abstract. The work presented in this paper is an attempt to bridge two co-existing realties: Semantic Web and Multi-Agent Systems. Semantic aware agents will be able to interoperate in a semantic way as well as to produce and consume semantically annotated information and services. Agents should be enhanced with tools and mechanisms in order to autonomously achieve these strategic and ambitious objectives. In this paper, we focus on what we consider the central issue when moving towards the vision of semantic multi-agent systems: the ontology management support. Due to the heterogeneity of resources available and roles played by different agents of a system, a one-level approach with the aim of being omni comprehensive seems to be seldom feasible. In our opinion, a good compromise is represented by a two-level approach: a light ontology management support embedded in each agent and one or more ontology servers, providing a more expressive and powerful support.

Keywords: Semantic web, multi-agent systems.

1 Introduction

One of the most important challenges in agent research is the realization of truly semantic aware agents, i.e. agents that are able to interoperate in a semantic way as well as to produce and consume semantically annotated information and services, supporting automated business transactions. To achieve this goal, researchers can take advantage of semantic Web technologies and, in particular, of OWL and its related software tools.

In this paper, we concentrate on what we consider the central theme when moving towards the vision of semantic multi-agent systems: the management and exploitation of OWL ontologies. We present a two-level approach, coping with the issues of managing complex ontologies and providing ontology management support to lightweight agents.

In the next section, we examine the rationale of embedding a light ontology support in each agent of a multi-agent system. Agents refer to this ontology support when they express the content of ACL messages, e.g. the domain concepts and the relationships that hold among them. Section 3 describes the implemented library providing agents with the aforementioned two-level ontology management support.

M. Kolp et al. (Eds.): AOIS 2005, LNAI 3529, pp. 140 – 153, 2006.

Finally, Section 4 gives some concluding remarks and presents our future research directions on ontology management in multi-agent systems.

2 A Perspective on Object-Oriented vs. OWL DL Model

The scenario in which our research is situated is characterized by different domain knowledge modelling techniques and by different needs. On the one hand, there is the semantic Web and OWL [1], the most recent development in standard ontology languages. On the other hand, the popularity of the Java language for the development of multi-agent systems pushes the need for having an ontology representation more in line with the object-oriented model.

The idea behind our two-level approach originates from the awareness that agents seldom need to deal with the whole complexity of a semantically annotated Web. Our objective is hence to cut off this complexity and provide each agent with simple artefacts to access structured information. These simple artefacts are based on Java technology.

At this point a crucial question arises: are the semantics implied by the object-oriented paradigm powerful enough? A comparison between the two models (object-oriented model, e.g. the Java data model, and OWL DL) is compelling in order to understand similarities and differences, and furthermore to evaluate the feasibility of using an object-oriented representation of the ontology. As a matter of fact, the language used to build an ontology influences the kind of details that one can express or takes into consideration.

Restricting only to the semantics of the object-oriented model, i.e. without considering the possibility of defining a meta-model, what we are able to express is a taxonomy among classes[1].

Briefly, we can rephrase the object-oriented model as follows. An instance of a class refers to an object of the corresponding class. Attributes are part of a class declaration. Objects are associated with attribute values describing properties of the object. An attribute has a name and a type specifying the domain of attribute values. All attributes of a class have distinct names. Attributes with the same name may, however, appear in different classes that are not related by generalization. Methods are part of a class definition and they are used to specify the behaviour and evolution of objects[2]. A generalization is a taxonomic relationship between two classes. This relationship specializes a general class into a more specific class. Generalization relationships form a hierarchy over the set of classes.

As far as OWL is concerned, it provides three increasingly expressive sublanguages designed for use by specific communities of implementers and users. Here we focus mainly on OWL DL (called simply OWL in the following), based on $\mathcal{SHIQ}$ Description Logics. OWL benefits from years of DL research and can rely on a well defined semantics, known reasoning algorithms and highly optimized implemented reasoners.

[1] We focus on the semantics of the so called "class based" model.

[2] The dynamic properties of the model are not dealt with in this paper, focussed on the structural aspects, even if they constitute an important part of the model.

OWL, as the majority of conceptual models, relies on an object centred view of the world. It allows three types of entities: concepts, which describe general concepts of things in the domain and are usually represented as sets; individuals, which are objects in the domain, and properties, which are relations between individuals.

At first glance OWL looks like an object-oriented model. Indeed, they are both based on the notion of class: in the object-oriented model, a class provides a common description for a set of objects sharing the same properties; in OWL, the extent of a class is a set of individuals.

Behind this resemblance, there is however a fundamental and significant difference between the two approaches, centred on the notion of property.

Individual attributes and relationships among individuals in OWL are called properties. The property notion appears superficially to be the same as the attribute/component in the object-oriented model. But, looking deeply to the DL semantics on which OWL DL is based, we can see that the two notions are fairly different. Formally [2], considering an interpretation I that consist of a set Δ^I (the domain of the interpretation) that is not empty and an interpretation function $\cdot^I$, to every atomic concept A is assigned a set $A^I \subseteq \Delta^I$ and to every atomic role R a binary relation $R \subseteq \Delta^I \times \Delta^I$. By means of the semantics of terminological axioms, we can make statements about how concepts and even roles are related to each other (e.g. $R^I \subseteq S^I$ inclusion relationship between two roles). What is clear is that roles in DL, and therefore OWL DL properties, are first-class modelling elements. Most of the information about the state of the world is captured in OWL by the interrelations between individuals. In other words, data are grouped around properties. For instance, all data regarding a given individual would usually be spread among different relations, each describing different properties of the same individual.

Differently, the object-oriented representation relies on the intentional notion of class, as an abstract data type (partially or fully) implemented [3], and on the extensional notion of object identifier. An object is strictly related and characterized by its own features including attributes and methods. In other words, data are grouped around objects, thought of as a collection of attributes/components.

As a consequence, in OWL it is possible to state assertions on properties that have no equivalent in the object-oriented semantics. Properties represent without any doubt one of the most problematic differences between OWL and object-oriented models.

To conclude, we can say that grounding the conceptual space of the ontological domain to a programming language such as Java has several obvious advantages but also some limitations. What we intend to do in next sub-section is an analysis of the weaknesses of the object-oriented representation compared to OWL, and to verify if its expressive power is powerful enough to capture the semantics of the agent knowledge base. In this study, we take into consideration that agents do not often need to face the computational complexity of performing inferences on large, distributed information sources; rather, they often simply need to produce and validate messages that refer to concepts of a given ontology.

2.1 Mapping OWL to Java

During the past years, much research work has been devoted to deal with the comparison between OWL and UML [4-5]. Among these, some considered the

mapping related to a particular object-oriented programming language: Java. Focussing on these, we can essentially identify two major directions followed by the research community in order to express the OWL semantics using the Java language.

1. The definition of a meta-model that closely reflect the OWL syntax and semantics. Examples are the modelling APIs of Jena [6-7] and OWL API [8-9]. The latter consists of a high-level programmatic interface for accessing and manipulating OWL ontologies. Its aim is to implement a highly reusable component suitable for applications like editors, annotation tools and query agents.
2. The use of the Java Beans API [10] to realize a complete mapping between the two meta-models. In particular, to cope with the central issue, i.e. the property-preserving transformation, [10] defines suitable *PropertyChecker* classes in order to support the semantics of the property axioms and restrictions. However, in our opinion, this approach lacks an explicit meta-model and therefore the corresponding explicit information. Moreover, it cannot be supported by a reasoner because of the impracticality of implementing one.

Our approach differs from those listed above since it aims at offering a two-level support: the most powerful one is based on Jena; the other is based on the object-oriented semantics.

When establishing a correspondence between two models, it is important to understand what the purpose of the mapping is. For example, the aim of having a full mapping and preserving the semantics is satisfied when using the Jena toolkit, whereas it is too strong in the case of the lightweight support. In the latter case, we decided to relax this constraint and consider a partial mapping, required only to be consistent (in the sense that it does not preserve semantics but only semantic equivalence [4]). This means that there is a one-to-one correspondence between instances of one model and the instances of the other model that preserves relationships between instances. This lets us use, for example, renaming and redundancy in order to achieve this goal, as in the use of interfaces in Java in order to express multiple inheritance.

For the sake of clarity and in order to avoid a lengthy dissertation, in the following we consider only the more salient aspects of the mapping, analysing commonalities as well as dissimilarities, and ending, in the successive sub-section, by delineating the application sphere of our approach.

Every OWL class is mapped into a Java interface containing the accessor method declarations (getters and setters) for properties of that class (properties whose domain is specified as this class). Then, for each interface, a Java class is generated, implementing the interface. Creating an interface and then separately implementing a Java class for each ontology class is necessary to overcome the single-inheritance limitation that applies to Java classes. In OWL, there is a distinction between named classes (i.e. primitive concepts), for which instances can only be declared explicitly, and defined classes (i.e. defined concepts), which specify necessary and sufficient conditions for membership. Java does not support this semantics and so only primitive concepts can be defined. In the following we refer only to named classes.

Individuals in OWL may be an instance of multiple classes, without one being necessarily a subclass of another. This is in contrast with the object-oriented model:

an object could get the properties of two classes only by means of a third one which has both of them in its ancestors. A workaround is thus to create a special subclass for this notion.

Considering the *terminological axioms* used to express how classes are related to each other, the only one that has an equivalent semantics in Java is the OWL synopsis *intersectionOf* (mapped as an interface which implements two interfaces). The *unionOf* OWL synopsis can be mapped in Java defining an interface as a super-interface of two interfaces but, in order to ensure the semantic equivalence, it is compulsory to prevent the implementation of the super-interface.

The constructs asserting completeness or disjointness of classes are those which characterized more OWL, from the point of view of the "open-world" assumption, i.e. modelling the state of the world with partial information. In OWL, classes are overlapping until disjointness axioms are entered. Moreover, generalization can be mutually exclusive, meaning that all the specific classes are mutually disjoint and/or complete, meaning that the union of the more specific classes completely covers the more general class. In Java, there is no way of expressing it and other similar properties (e.g. *equivalentClass*); the representation of the world that we can state using this model can only refer to a "closed-world" assumption. This obviously constitutes a limitation when one cannot assume that the knowledge in the knowledge base is complete.

Regards properties, since they are not first-class modelling elements in Java, it is not possible to create property hierarchies and to state that a property is symmetric, transitive, equivalent or the inverse of another property. Properties can be used to state relationships between individuals (*ObjectProperty*) or from individuals to data values (*DatatypeProperty*). DatatypeProperties can be directly mapped into Java attributes of the corresponding data type and ObjectProperties to Java attributes whose type is the class specified in the property's range. In OWL there are constraints that can be enforced on properties:

1. Cardinality constraints state the minimum and maximum number of objects that can be related;
2. The "domain" constraint limits the individuals to which the properties can be applied;
3. The "range" constraint limits the individuals that the property may have as its value.

Java accessor methods could ensure that cardinality constraints be satisfied. This information, however, is implicit and embedded in the class source code and it would not become known to a possible reasoner and therefore it would be most likely of no use.

Concerning the domain restriction, if the property domain is specified as a single class, the corresponding Java interface contains declarations of accessor methods for the property. In the case of a multiple domain property, there are two possible alternatives:

1. The domain is an *intersection-of* all the classes specified as the domain; to cope with this we create an intersection interface (see above).
2. Multiple alternative domains are defined using the *unionOf* operator; we can cope with this creating a union interface but with the limitations expressed above.

Finally, in relation to the range restriction, our approach fails to account for multi-range properties, since variables in Java can be only of one type.

It clearly emerges, from the previous analysis, that the Java language expressiveness is lower even than OWL Lite but, despite this, in our view, it is still valuable with respect to the common agent needs.

2.2 Reasoning About Knowledge

Although DLs (and hence OWL DL) and object-oriented models have a common root in class-based models, they were developed by different communities and for different purposes. The different target applications significantly affect the expressiveness of the languages and consequently the reasoning services that can be performed on the corresponding knowledge base.

The object model only permits the specification of necessary conditions for the class (i.e. the definition of the properties that must be owned by objects belonging to a specific class) that are not sufficient to identify a member of the class. The only way to associate an instance to a class is therefore to explicit assert its membership. As a consequence some basic reasoning services lose their importance and significance (e.g. knowledge base consistency, subsumption and instance checking). A fairly common complex reasoning service, i.e. classification, also plays a marginal role in an object-oriented model. In fact, in DL, the terminological classification consists in making explicit the taxonomy entailed by the knowledge base. Whereas the classification of individuals has its role in DL, since individuals can be defined giving a set of their properties and therefore objects' classes, membership can be dynamically inherited.

The previous remarks lead us to consider the aspect that differentiates even more between the two models, that is the divergent assumption on the knowledge about the domain being represented - open vs. closed world assumption. Indeed while a DL-based system contains implicit knowledge that can be made explicit through inference, a system based on an object-oriented model exhibits a limited use of entailment. Inheritance may represent a simple way of expressing implicit knowledge (a class inherits all the properties of its parents without explicit specifying it). Another way is to represent part of the information within methods (e.g. initialization methods), but this implicit information is not (or hard) available to a potential reasoner.

If we consider the knowledge base as a means of storing information about individuals, an interesting complex reasoning task is represented by retrieval. Retrieval (or query answering) consists in finding all the individuals in the knowledge base in a concept expression. The information retrieval task plays a leading role in a knowledge base centred on an object-oriented representation.

3 System Architecture

The concrete implementation of the proposed system is a direct result of the evaluations set out in the previous sections. The proposed two-level approach to

ontology management is implemented as a framework providing the following functionality:

1. Light support: to import OWL ontologies as an object-oriented hierarchy of classes;
2. Ontology Server: to provide the centralized management of shared ontologies.

3.1 OWLBeans

The OWLBeans framework, which is going to be presented in this section, does not deal with the whole complexity of a semantically annotated Web. Instead, its purpose is precisely to cut off this complexity, and to provide simple artefacts to access structured information.

In general, interfacing agents with the Semantic Web implies the deployment of an inference engine or of a theorem prover. In fact, this is the approach we are currently following to implement an agent-based server to manage OWL ontologies. Instead, in many cases, autonomous agents cannot (or do not need to) face the computational complexity of performing inferences on large, distributed information sources. The OWLBeans framework is mainly thought for these agents, for which an object-oriented view of the application domain is enough to complete their tasks.

The software artefacts produced by the framework, i.e. mainly JavaBeans and simple metadata representations used by JADE [11], are not so expressive as OWL-DL. But in some context this is not required. Conversely, especially if software and hardware resources are very limited, it is often preferable to deal only with common Java interfaces, classes, attributes and objects. Its main functionality is to extract a subset of the relations expressed in an OWL document for generating a hierarchy of JavaBeans, and possibly for creating a corresponding JADE ontology to represent metadata. However, given its modular architecture, it also provides other functionality, e.g. to save a JADE ontology into an OWL file or to generate a package of JavaBeans from the description provided by a JADE ontology.

Intermediate ontology model. In order to keep the code maintainable and modular, we decided to base the framework on an internal, intermediate representation of the ontology. This intermediate model can be alternatively used to generate the sources of some Java classes, a JADE ontology or an OWL file. The intermediate model itself can be filled with data obtained, e.g. by reading an OWL file or by inspecting a JADE ontology.

The main design goals of the internal ontology representation were:

1. *Simplicity*: it had to include only few simple classes to allow a fast and easy introspection of the ontology. The model had to be simple enough to be managed in scripts and templates; in fact, one of the main design goals was to have a model be directly used by a template engine to generate the code.
2. *Expressiveness*: it had to include the information needed to generate JavaBeans and all other desired artefacts. The main guideline was to avoid limiting the translation process. The intermediate model had to be as simple as possible, though not creating a metadata bottleneck in the translation of an OWL ontology to JavaBeans.

3. *Primitive data-types*: it had to handle not only classes, but even primitive data-types, since both Java and OWL classes can have properties using primitive data-types as their range.
4. *External references*: ontologies are often built extending more general classifications and taxonomies. For example, an ontology can detail the description of some products in the context of a more general trade ontology. We wanted our model not to be limited to single ontologies, but to allow the representation of external entities too: classes may extend other classes, defined locally or in other ontologies, and property ranges may allow not only primitive data-types and internal classes, but also classes defined in external ontologies.

One of the main issues related to properties, since they are handled in different ways in description logics and in object-oriented systems (see the previous section). For the particular aims and scope of OWLBeans, property names must be unique only in the scope of their own class in object-oriented systems, while they have global scope in description logics. Our choice, in the internal model design, was to have properties "owned" by classes. This allows an easier manipulation of the meta-objects while generating the code for the JavaBeans, and a more immediate mapping of internal description of classes to the desired output artefacts.

The intermediate model designed for the OWLBeans framework is made of just a few, very simple classes. The simple UML class diagram shown in Fig. 1 describes the main classes of the intermediate model package.

The root class is *OwlResource*, which is extended by all the others. It has just two fields: a local name and a namespace, which are intended to store the same data as resources defined in OWL files. All the resources of the model – refernces, ontologies, classes and properties – are implicitly *OwlResource* objects.

OwlReference is used as a simple reference, to point to super-classes, range and domain types, and does not add anything to the *OwlResource* class definition. It is defined to underline the fact that classes cannot be used directly as ranges, domain or parents.

OwlOntology is nothing more than a container for classes. It owns a list of *OwlClass* objects. It inherits from *OwlResource* the *name* and *namespace* fields. In this case the namespace is mandatory and is supposed to be the namespace of all local resources, for which it is optional.

OwlClass represents OWL classes. It points to a list of parents, or super-classes, and owns a list of properties. Each parent in the list is an *OwlReference* object, i.e. a name and a namespace, and not an *OwlClass* object. Its name must be searched in the owner ontology to get the real *OwlClass* object. Properties instead are owned by the *OwlClass* object, and are stored in the properties list as instances of the *OwlProperty* class.

OwlProperty is the class representing OWL properties. As in UML, their name is supposed to be unique only in the scope of their "owner" class. Each property points to a domain class and to a range class or data-type. Both these fields are simple *OwlReference* objects: while the first contains the name of the owner class, the latter can indicate an *OwlClass*, or an XML data-type, according to the namespace. Two more fields are present in this class: *minCardinality* and *maxCardinality*. They are

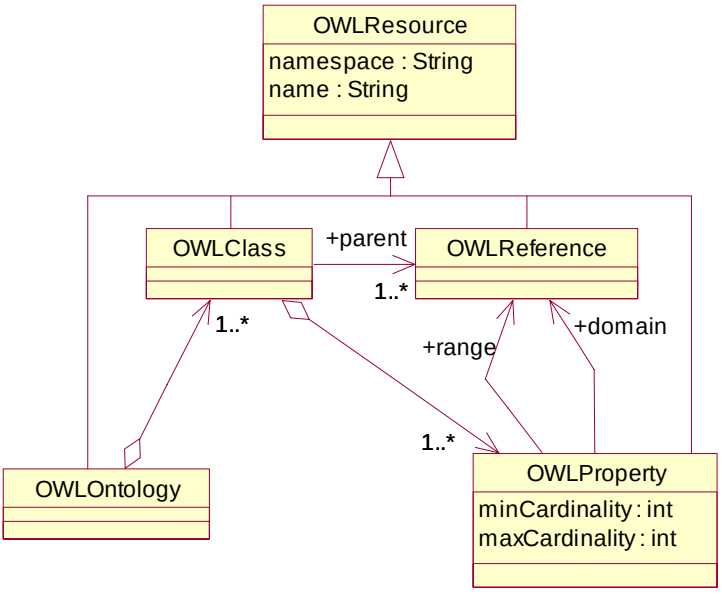

Fig. 1. Class diagram of the intermediate model

used to store respectively the minimum and maximum allowed cardinality for the property values. A *minCardinality* = 0 has the implicit meaning of an optional property, while *maxCardinality* = 1 has the implicit meaning of a functional property.

It is worth pointing the unusual treatment of indirect references to *OwlClass* objects. This decision has two main advantages over direct Java references to final objects. Parsing an OWL file is a bit simpler, since references can point to classes that are not yet defined. Furthermore, in this way, super-classes, domain and ranges are not forced to be local classes, but can be references to resources defined somewhere else.

In our framework, the intermediate model is used as the glue to put together the various components needed to perform the desired, customizable task. These components are classes implementing the *OwlReader* or the *OwlWriter* interface, representing ontology readers and writers, respectively. While readers can read an intermediate representation of the ontology, acquiring metadata from different kinds of sources, writers, instead, can use this model to produce the desired artefacts.

The current version of the framework provides readers to inspect OWL files and JADE ontologies, and writers to generate OWL files, source files of JavaBeans and JADE ontologies.

Reading OWL Ontologies. Two classes are provided to manage OWL files. *OwlFileReader* allows reading an intermediate model from an OWL file, while *OwlFileWriter* allows saving an intermediate model to an OWL file. These two classes respectively implement the *OwlReader* and *OwlWriter* interfaces and are defined in the package confining all the dependencies from the Jena toolkit.

The direct process, i.e. converting an OWL ontology into the intermediate representation, is possible only under quite restrictive limitations, mainly caused by

the rather strong differences between the OWL data model and the object-oriented model. In fact, only few, basic features of the OWL language are supported.

Basically, the OWL ontology is first read into a Jena *OntModel* object and then all classes are analysed. In this step, all anonymous classes are just discarded. For each one of the remaining classes, a corresponding *OwlClass* object is created in the internal representation. Then, all properties listing the class directly in their domain are added to the intermediate model as *OwlProperty* objects. Here, each defined property points to a single class as domain and to a single class or data-type as range. Set of classes are not actually supported. Data-type properties are distinguished in our model by the namespace of their range: *http://www.w3.org/2001/XMLSchema#*. The only handled restrictions are *owl:cardinality*, *owl:minCardinality* and *owl:maxCardinality*, which are used to set the *minCardinality* and *maxCardinality* fields of the new *OwlProperty* object. The *rdfs:subClassOf* element is handled in a similar way: only parents being simple classes are taken into consideration and added to the model.

All remaining information in the OWL file is lost in the translation, as it does not fit into the desired object-oriented model.

Generating JavaBeans. Rather than generating the source files of the desired JavaBeans directly from the application code, we decided to integrate a template engine in our project. This helped to keep the templates out of the application code, and centralized in specific files, where they can be analysed and debugged much more easily. Moreover, new templates can be added and existing ones can be customized without modifying the application code.

The chosen template engine was Velocity [12], distributed under LGPL licence by the Apache Group. It is an open source project with a widespread group of users. While its fame mainly comes from being integrated into the Turbine Web framework, where it is often preferred to other available technologies, as JSP pages, it can be effortlessly integrated in custom applications, too.

Currently, the OWLBeans framework provides templates to generate the source file for JavaBeans and JADE ontologies. JavaBeans are generated according to the mapping between classes and concepts that we described in the previous sections. In particular, all JavaBeans are organized in a common package where, first of all, some interfaces mapping the classes defined in the ontology are written. Then, for each interface, a Java class is generated, implementing the interface and all accessor methods needed to get or set properties.

As stated in Section 2, creating an interface and then a separate implementing Java class for each ontology class is necessary to overcome the single-inheritance limitation that applies to Java classes.

The generated JADE ontology file can be compiled and used to import an OWL ontology into JADE, thus allowing agents to communicate about the concepts defined in the ontology. The JavaBeans will be automatically marshalled and un-marshalled from ACL messages in a completely transparent way.

Additional components. Additional components are provided to read and write ontologies in different formats.

For example, the *JadeReader* class allows the loading of a JADE ontology and saving it in OWL format or generating the corresponding JavaBeans.

Another component is provided to instantiate an empty JADE ontology at run time, and to populate it with classes and properties read from an OWL file or from other supported sources. This proves useful when the agent does not really need JavaBeans but can use the internal ontology model of JADE to manage the content of semantically annotated messages.

Finally, the *OwlWriter* class allows an ontology to be converted from its intermediate representation to an OWL model. This is quite straightforward, since all the information stored in the intermediate model can easily fit into an OWL ontology, in particular into a Jena *OntModel* object. One particular point deserves attention. While the property names in the OWLBeans model are defined in the scope of their owner class, all OWL properties are instead first level elements and share the same namespace. This poses serious problems if two or more classes own properties with the same name and, above all, if these properties have different ranges or cardinality restrictions.

In the first version of the OWLBeans framework, this issue is faced in two ways: if a property is defined by two or more classes, then a complex domain is created in the OWL ontology for it; in particular, the domain is defined as the union of all the classes that share the property, using an *owl:UnionClass* element. Cardinality restrictions are specific to classes in both models and are not an issue. Currently, the range is assigned to the property by the first class that defines it and is kept constant for the other classes in the domain. Obviously this could be incorrect in some cases. Using some class-scoped owl*:allValuesFrom* restrictions could solve most of the problems, but difficulties would arise in the case of a property defined in some classes as a data-type property and somewhere else as an object property.

Another mechanism allows the optional use of the class name as a prefix for the names of all its properties, hence automatically enforcing different names for properties defined in different classes. This solution is appropriate only for ontologies where property names can be decided arbitrarily. Moreover, it is appropriate when resulting OWL ontologies are used only to generate JavaBeans and JADE ontologies, since in this case the leading class name would be automatically stripped off by the *OwlFileReader* class.

Scripting Engine. The possibilities opened by embedding a scripting engine into an agent system are various. For example, agents for e-commerce often need to trade goods and services described by a number of different, custom ontologies. This happens in the Agentcities network [13], where different basic services can be composed dynamically to create new compound services.

To increase adaptability, these agents should be able to load needed classes and code at runtime. The OWLBeans package allows them to load into the Java Virtual Machine some JavaBeans directly from an OWL file, together with the ontology-specific code needed to reason about the new concepts.

This is achieved by embedding *Janino* [14], a Java scripting engine, into the framework. Janino can be used as a special class loader capable of loading classes directly from Java source files without first compiling them into bytecode.

Obviously, pre-compiled application code cannot access newly loaded classes, which are not supposed to be known at compile time. However, the same embedded scripting engine can be used to interpret some ontology specific code, which could be loaded at run time from the same trusted source of the OWL ontology file, for example, or provided to the application in other ways.

3.2 Ontology Server

The OWLBeans framework allows agents to import taxonomies and classifications from OWL ontologies, in the form of an hierarchy of Java classes. Clearly, a more general solution must be provided for all those cases where a simplified, object-oriented view of the ontology is not enough.

For all those applications, that need a complete support of OWL ontologies, we are developing an Ontology Server. It is an agent-based application providing ontology knowledge and reasoning facilities for a community of agents. The main rationale for building on Ontology Server is to endow a community of agents with the ability to automatically process semantically annotated documents and messages. The Ontology server shares a common knowledge base about some application domains with this community of agents.

The first functionality is related to loading, importing, removing ontologies. Apart from loading ontologies at agent startup, specific actions are defined in terms of ACL requests to add ontologies to the agent knowledge base, and to remove them. Ontologies that are linked through import statements can be loaded automatically with a single request. Moreover, new relations among ontologies can be dynamically created, and existing ones can be destroyed. This import mechanism can be used to build a distributed knowledge base hierarchy; in this way, a new ontology can be plugged in easily and inherit the needed general knowledge base, instead of building it totally from scratch.

After the initial set-up, through a number of potentially related ontologies, this knowledge base can be queried from other agents. A set of predicates is defined, to check the existence of specific relations among entities. For example the Ontology Server can be asked about the equivalence of two classes or about their hierarchical relationships.

Apart from checking the existence of specific relations, the knowledge base can also be used to search for the entities satisfying certain constraints. For example, the list of all the super-classes, or of all the sub-classes, of a given class can be obtained.

Finally, client agents may be allowed to modify an ontology managed by the Ontology Server. Agents can ask to add new classes, individuals and properties to the ontology or to remove defined entities. Moreover, relations among ontology entities can also be added and removed at runtime.

Our current implementation is built as a JADE agent, using the Jena toolkit to load and manage OWL ontologies. An inference engine can be plugged into the application to reason on the knowledge base. An ontology is defined, to allow the management of the internal knowledge base. ACL requests, to access and query the Ontology Server about its knowledge base, can use this meta-ontology to represent their semantic content.

As a final point, for the Ontology Server to be really useful in an open environment, we are adding proper authorization mechanisms. In particular, we are leveraging the underlying JADE security support to implement a certificate-based access control. Only authenticated and authorized users will be granted access to managed ontologies. The delegation mechanisms of JADE allow the creation of communities of trusted users, which can share a common ontology, centrally managed by the Ontology Server.

Finally, we are developing a graphical user interface to allow the interaction with the Ontology Server through Web pages. It allows both the introspection of the existing knowledge base, as well as its modification by human users.

4 Conclusion

In this paper, we have presented a software implementation intended to provide an OWL ontology management support for multi-agent systems implemented by using JADE. The key feature that distinguishes our approach from others is the fact that lightweight agents have the possibility of directly managing ontologies that can be mapped in JavaBeans, while they can take advantage of special agents, called Ontology Servers, when they need to use more complex. Well aware of the need to clearly define the weakness of our approach in comparison to a fully-fledged OWL support, we have carried out a meticulous analysis of its expressiveness.

Our current activities are related to the experimentation of the implemented software in the realization of a multi-agent system for the remote assistance of software programmers. Furthermore, we are working on its improvement by trying alternative solutions to the use of the Jena software tool.

References

1. Word Wide Web Consortium (W3C). OWL. Web Ontology Language. http://www.w3.org/TR/owl-ref.
2. The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press (2002)
3. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, 2nd edition (1997)
4. Baclawski, K., Kokar, M.K., Kogut, P., Hart, L., Smith, J.E., Letkowski, J., Emery, P.: Extending the Unified Modeling Language for ontology development. International Journal Software and Systems Modeling (SoSyM) 1(2) (2002) 142-156
5. Hart, L., Emery, P., Colomb, B., Raymond, K., Taraporewalla, S., Chang, D., Ye, Y., Kendall, E., Dutra, M.: OWL Full and UML 2.0 Compared (2004). http://www.omg.org/docs/ontology/04-03-01.pdf.
6. Carroll, J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: Implementing the Semantic Web Recommendations. In Proc 13th Int World Wide Web Conference, New York, NY (2004) 74-83
7. Jena, HP Labs Semantic Web Toolkit software and documentation. http://jena.sourceforge. net/.
8. Bechhofer, Volz, R., Lord, P.: Cooking the Semantic Web with the OWL API. In Proc. Intl Semantic Web Conference, Sanibel Island, FL, USA (2003) 659-675

9. OWL API software and documentation. http://owl.man.ac.uk/api.shtml.
10. Kalyanpur, A., Pastor, D., Battle, S., Padget, J.: Automatic Mapping of Owl Ontologies into Java. In Proceedings of Software Engineering .and Knowledge Engineering Conference. (SEKE) 2004, Banff, Canada (2004)
11. JADE software and documentation. Available at http://jade.tilab.com.
12. Velocity software and documentation. Available at http://jakarta.apache.org/velocity.
13. The Agentcities Network project home page. http://www.agentcities.net.
14. Janino software and documentation. Available at http://janino.net.

AOSE and Organic Computing - How Can They Benefit from Each Other?

Position Paper

Bernhard Bauer and Holger Kasinger

University of Augsburg, 86135 Augsburg, Germany
{bauer, kasinger}@informatik.uni-augsburg.de

Abstract. Organic Computing (OC) is an upcoming research area with strong relationships to the ideas and concepts of agent-based systems. Therefore, in this paper we will have a closer look at agent systems, organic computing systems (as well as autonomic computing systems) and state commonalities and examine divergences between them. We then propose a common view on these technologies and show how they can benefit from each other with regard to software engineering (SE).

1 Introduction

Over the past few years technical systems such as aeroplanes, vehicles, telecommunication networks and manufacturing installations have become more and more complex. This is not only a result of the continuing evolution in microelectronics but also of the immense embedding of huge hardware and software complexes into these systems. But the producers' painful experiences show that these systems today are already very difficult to manage. Thus, with respect to future evolution, new advanced management principles have to be developed. A feasible principle is an autonomic behaviour of the systems, addressed by two research directions: namely agent technology and Organic / Autonomic Computing (AC).

Agent technology is believed to be able to play a key role in this "revolution", e.g. by automating daily processes, enriching higher level communication or enabling intelligent service provision. An intelligent agent is "a computer system, situated in some environment that is capable of flexible autonomous actions in order to meet its design objectives" [1]. The real strength of agents is based on the community of a multi-agent system (MAS) and negotiation mechanisms and coordination facilities (see [2] for more details). An MAS is "a dynamic federation of software agents that are coupled through shared environments, goals or plans and that cooperate and coordinate their actions" [3]. It is this ability to migrate, communicate, coordinate and cooperate that makes agents and multi-agent systems a worthwhile metaphor in computing and that makes them attractive when it comes to tackling some of the requirements in next-generation systems.

M. Kolp et al. (Eds.): AOIS 2005, LNAI 3529, pp. 154–167, 2006.

Another worthwhile metaphor is provided by OC systems [4] that can be considered as an extension to AC systems [5]. The latter — driven by IBM since 2001 — draw analogies from the human body, in particular from the autonomic nervous system, where all reactions occur without explicit override by the human brain — so to say autonomous. By embedding this behaviour into technical systems, the administrative complexity of next-generation systems can be left to the systems themselves. IBM refers to this autonomy as "self-management", which includes four so-called "self-* properties", namely self-configuration (configuration and reconfiguration according to policies), self-optimization (permanent improvement of performance and efficiency), self-healing (reactive and proactive detection, diagnostics and reparation of localized SW/HW-problems) and self-protection (defence of the system as a whole). Furthermore, AC systems are self-aware, context-sensitive, non-proprietary, anticipative and adaptive. OC systems instead draw analogies from the biological world and try to use perceptions about the functionality of living systems for the development and management of artificial and technical systems respectively. In addition to the properties of AC systems they are defined as being self-organizing (hence they do not necessarily have to be self-aware).

As OC systems basically have the same objectives and concepts as AC systems, we will treat them mostly as one single technology for the rest of this paper, which is organized as follows: In Section 2 we present the concepts of agents as well as AC/OC and the existing SE approaches for these technologies. Section 3 relates the technologies and presents a common view on them. Based on this view, in Section 4, we present a development process, which helps to benefit agent-oriented software engineering (AOSE) and OC from each other. Section 5 presents a short case study that exemplifies a couple of process activities before we conclude with open issues and an outlook for further research in Section 6.

2 Concepts

In this section we give an overview on agent technology as well as on AC/OC and consider the associated software engineering methodologies.

2.1 Agents

Software agents are software components characterized by **autonomy** (to act on their own), **reactiveness** (to process external events), **proactiveness** (to reach goals), **cooperation** (to efficiently and effectively solve in common tasks), **adaptation** (to learn by experience) and **mobility** (migration to new places). For further details on agent technology see e.g. [6] or [7].

Often, agents are subdivided into three functional sections (see Fig. 1): The **agent body** wraps a software component (e.g. a database, a calendar or an external service) and controls it through the software API. Connected to external software, the agent acts as an application agent by transforming the application API into agent communication language (ACL) and vice versa. Messages of

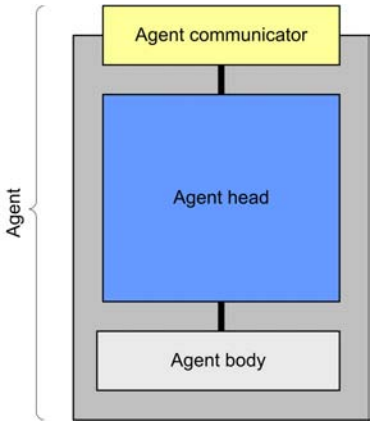

Fig. 1. Logical structure of an agent

such ACLs are highly structured and must satisfy standardized communicative (speech) acts which define the type and the content of the messages (like FIPA-ACL [8] or KQML [9]). The order of exchanged messages is fixed in protocols according to the relation of agents or the intention of the communication.

The **agent head** is responsible for the agent's intelligence. It is connected to the agent body on one side and to the agent communicator on the other. The agent head contains knowledge bases storing knowledge of certain types like facts, beliefs, goals or intentions, preferences, motivations and desires concerning the agent itself or associated ones. Furthermore, it contains a world model as an abstraction of relevant states of the real world. It is updated by information from other agents or through real world interfaces, e.g. sensors. The agent head is able to evaluate incoming messages with respect to its goals, plans, tasks, preferences and to the world model.

The **agent communicator** converts logical agent addresses into physical addresses and delivers messages on behalf of the agent head through appropriate channels to the receivers. Furthermore, the communicator listens for incoming messages (e.g. by running an event loop) and forwards them to the agent head. The agent behaviour should be benevolent, which means that an agent at least understands the interaction protocols and reacts accordingly.

Beyond these mobile and cooperating agents in the literature additional kinds of agents can be found: **Search agents** are, e. g., search engines which scan the WWW to store the information in local databases in order to allow efficient keyword search. **User agents** are, e. g., Microsoft Office agents supporting the user during work with the product and give some predefined information to the user according to his/her interaction.

2.2 Autonomic/Organic Computing

According to [10], AC systems are composed of four levels: on the lowest level **managed resources (MR)**, e.g. HW/SW-components as servers, databases

or business applications, are located, together making up the complete IT infrastructure. So-called **touchpoints** on the next level provide a manageability interface — similar to an API — for each MR by mapping standard sensor and effector interfaces on the sensor and effector mechanisms (e.g. commands, configuration files, events or log files) of a specific MR. The next level is composed of so-called **touchpoint autonomic managers (TAM)** directly collaborating with the MRs and managing them through their touchpoints.

Generally, an **autonomic manager (AM)** implements an intelligent control loop (closed feedback loop) called a **MAPE loop**. The latter is composed of the components **monitor** (collects, aggregates, filters and reports the MR's details), **analyse** (correlates and models complex situations), **plan** (constructs actions needed to achieve goals) and **execute** (controls execution of a plan). Additionally, a knowledge component provides the data used by the four components, including policies, historical logs and metrics. Together with one or more MRs, an AM represents an **autonomic element (AE)** (see Fig. 2). A TAM also provides a sensor and an effector to **orchestrating autonomic managers (OAM)** residing on the top level. The latter achieve system-wide autonomic behaviour, since TAMs are only able to achieve autonomic behaviour for their controlled MRs.

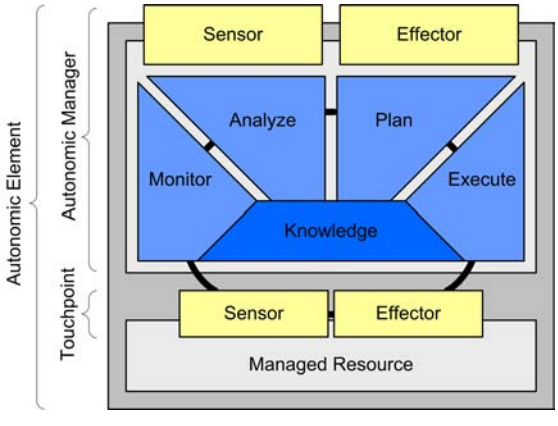

Fig. 2. Logical structure of an autonomic element

As (strong) self-organizing systems (like OC systems) are defined as systems "that change their organization without any explicit — internal or external — central control" [11], there can be no single instance within an OC system that is aware of all system's components or states. From our point of view, system-wide autonomic behaviour in OC systems is in contrast to AC systems leading to emergent behaviour of the system's component interactions and not the achievement of a single OAM. This issue has significant impact on software engineering but not on the concepts mentioned above, which are also used in OC systems.

2.3 Software Engineering Methodologies

Agent-oriented Software Engineering Methodologies An extensive number of AOSE methodologies and tools are available today (see our work in [12] or [13] for a more detailed survey), and the agent community is facing the problem of identifying a common vocabulary to support them.

The knowledge engineering community inspired most early approaches supporting the SE of agent-based systems: CommonKADS [14] was developed to support knowledge engineers in modelling expert knowledge and developing design specifications in textual or diagrammatic form. To consider agent-specific aspects CoMoMAS [15] and MAS-CommonKADS [16] were developed.

Gaia [17] is a methodology designed to deal with coarse-grained computational systems, having static organization structures and agents with static abilities and services. ROADMAP [18] extends Gaia by adding elements to deal with requirements analysis in more detail by using use cases, handling open system environments and specification of interactions. SODA [19] addresses aspects like open systems or self-interested agents, based on the analysis and design of agent societies (exhibiting global (emergent) behaviour not deducible from the behaviour of the individual agents) and agent environments.

One of the first methodologies for the development of BDI agents based on OO technologies was presented in [14] and [20]. The methodology distinguishes between the external viewpoint — the system is decomposed into agents, modelled as complex objects characterized by their purpose, their responsibilities, the services they perform, the information they require and maintain, and their external interactions — and the internal viewpoint — the elements required by a particular agent architecture must be modelled for each agent, i.e. an agent's beliefs, goals and plans.

MESSAGE [21] is a methodology that extends UML by agent-related concepts (inspired e.g. by Gaia). TROPOS [22] uses UML for the development of BDI agents. Prometheus [23] it is an iterative methodology covering the complete SE process and aiming at the development of intelligent agents using goals, beliefs, plans and events, resulting in a specification that can, for example, be implemented with JACK [24]. MaSE [25] has been developed to support the complete software development life cycle. PASSI [26] is an agent-oriented iterative requirement-to-code methodology for the design of multi-agent systems mainly driven from experiments in robotics.

Autonomic / Organic Computing Methodologies. Continuous and consistent SE methodologies for AC/OC systems are more or less unavailable nowadays, since most of the research activities are in the area of algorithms, middleware, hardware concepts as well as application areas. Nevertheless, the objective in particular of OC has to be on the control of such systems by engineering methods. Traditional SE methods are strictly hierarchical and follow a top-down approach by transforming the entire specification into detailed modules. For emergent and self-organizing systems, this strict approach is abandoned. System states have to be reached that are not imagined beforehand. This is a

fundamental contradiction between a top-down-control and a creative bottom-up-behaviour.

Today it is not clear how to combine these opposite tendencies. However, there are some approaches (see e. g. [27]) based on an industry-ready software engineering process (Unified Process) as well as approaches based on constraint propagation, the use of assertions and so-called observer/controller architectures. Assertions can be used for monitoring values of special variables. Yet, the limitation of emergent behaviour of OC systems will be crucial for their technical application. Thus, constraints play an important role to the limitation of learning in self-organizing systems as constraint violations result in warnings.

3 Relating Agents and Organic Computing

Based on the presented concepts we try to relate agents and OC in this section and propose a common view on these technologies.

Both technologies incorporate managed objects, either software components wrapped in the agent body or managed resources on the OC-side. In addition, both technologies have an institution for intelligent and autonomic behaviour, namely the agent head and the autonomic manager respectively. Moreover, an agent communicator is in some sense comparable to a touchpoint in OC.

Thus, in order to bring the technologies together, we consider an autonomic element from now on as the combination of agents and organic computing with the following properties: having a BDI mental model about other autonomic elements; using a MAPE loop similar to the control loop of agents, with monitoring and analyzing the environment and messages, consulting the knowledge base, planning and execution; managing the internal behaviour automatically, like OC does it, without interaction with the environment; interacting with its environment, not only via direct messages but also via e.g. stigmergy — therefore the environment has to be modelled explicitly, as for swarm intelligence or ant algorithms. Moreover, an autonomic element community consists of cooperating autonomic elements explicitly communicating based on speech acts and interaction protocols or implicitly via the environment. Additionally these cooperating elements have to satisfy global system rules such that no unintentional behaviour of the system takes place.

Having this in mind, we propose a metamodel for both an MAS with OC properties and OC systems as MASs (see Fig. 3). Therefore, we have combined different proven concepts of existing agent architectures and their SE methodologies as well as AC/OC concepts.

Similar to many existing agent methodologies, a **role** is the central architectural concept. The complete set of roles builds up the **environment**. The life cycle of a role is traditionally described as follows. A role or rather the enacting agent recognizes a situation, makes a decision based upon it and executes appropriate activities. The recognition of situations is based on **events**. **Regular events** are familiar to a role, e.g. by design or by adaption, whereas **irregular events** are new to a role, e.g. by failure appearance. **Norms** regulate the

behaviour of a role and are a generalization of either a **permission**, an **obligation** or a **prohibition** and consist of a goal and activation as well as deactivation events. The decision making is based on **plans** that fire certain events at the end (as notification of being in a certain state), which may correspond to a norm's goal or event respectively. A plan consists of actions (internal activities of a role) and interactions (external activities between different roles) and are chosen accordingly to a goal of an activated norm. **Interactions** are implemented by specific **interaction protocols**. The relation between interactions and interaction protocols is the same as between interfaces and their implementations. Thus, according to diverse requirements, an interaction may be implemented by different kinds of protocols for direct (e.g. by auctions) or indirect (e.g. by stigmergy) communication. Interactions and actions are both implemented by **services** with different visibilities.

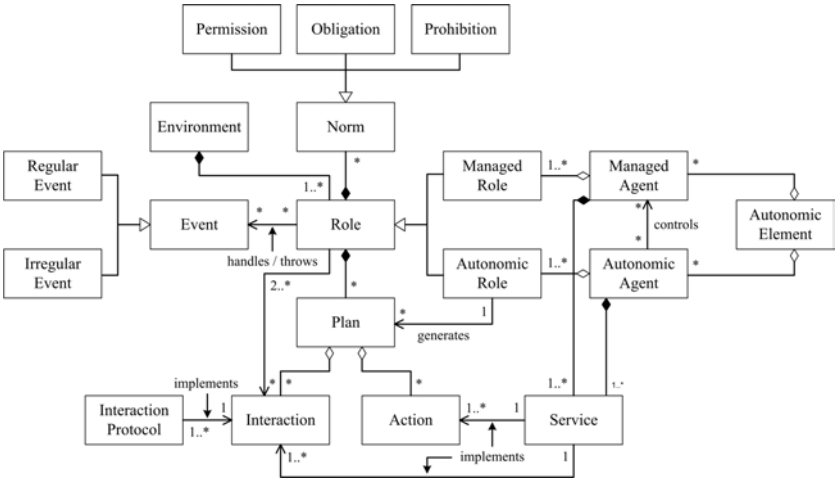

Fig. 3. The metamodel for organic computing systems

Roles are logically divided into **managed roles (MR)** and **autonomic roles (AR)** (similar to the AC concepts). MRs are responsible for the business logic of a system and reside on versatile resources. They are controlled by one or more ARs that are responsible for the self-management of a system. ARs do not necessarily have to be located at the same resource as its MRs. In contrast to MRs, the ARs are able to generate new plans based on the received data of their MRs. The latter do not have to generate new plans as they communicate the occurrence of irregular events to their monitoring ARs and mostly are not in possession of further required information. Both roles are taken over dynamically by **managed agents** and **autonomic agents** respectively. **Autonomic elements** contain one or more autonomic agents and managed agents at the same time.

4 Software Engineering for OC and AO Systems

As a result of the common view presented in the previous section, we propose a development process in this section which can be used for both AOSE and OC. Although the process is in early stages and more details will be forthcoming, the general way of development already becomes clear. The process is based on the Model Driven Architecture (MDA), a framework for software development driven by the Object Management Group (OMG). It comprises a **Computation Independent Model (CIM)** (model of a system that abstracts from any computation), a **Platform Independent Model (PIM)** (model of a system that abstracts from any specific platform) and a **Platform Specific Model (PSM)** (model of a system that is tailored to one or more specific implementation platforms). For a more detailed description see [28].

The process consists of 19 activities and encompasses an analysis phase (activities 1-5) and a design phase (activities 6-19). Each activity results in a specific model either in the CIM (analysis phase) or the PIM (design phase) (see Fig. 4). An implementation phase is not considered yet, but can be added smoothly in the future. Notice, the process does not prescribe a process model.

The analysis phase consists of the activities (1) 'Definition of the business context', (2) 'Definition of business processes being supported', (3) 'Characterization of the environment', (4) 'Assembly of potential use cases' and (5) 'Assembly of common vocabulary'. The resulting models are: **Business Context Model:** As a result of (1) the business context of the future system is modelled by a UML activity diagram. This model only considers higher level correlations and abstracts from concrete business processes; **Business Process Model:** As a result of (2) the business processes supported by the latter system are modelled by a UML activity diagram; **Environment Model:** As a result of (3) important environment objects of all types are modelled by a UML class diagram; **Use Case Model:** As a result of (4) the system application is declared abstractly in a UML use case diagram. The model is supported by a UML sequence diagram to explain the message flow of the system clearly; **Ontology Model:** As a result of (5) all important knowledge blocks and common vocabulary are categorized in a UML class model.

The design phase consists of the activities (6) 'Identification of MRs', (7) 'Specification of norms for MRs', (8) 'Development of plans for MRs', (9) 'Derivation of interactions between MRs', (10) 'Specification of services of MRs', (11) 'Identification of ARs', (12) 'Specification of norms for ARs', (13) 'Development of an analysis for ARs', (14) 'Development of plans for ARs', (15) 'Derivation of interactions between ARs', (16) 'Specification of services of ARs', (17) 'Development of interaction protocols', (18) 'Identification of AE' and (19) 'Deployment of AE'. The resulting models of this phase are: **Managed Role Model:** As a result of (6) the MRs are identified and modelled similar to a class in a UML composition structure diagram; **MR Norm Model:** As a result of (7) the norms (containing goals, activation and deactivation events) of MRs are specified and modelled similar to a class in a UML class model; **MR Plan Model:** As a result of (8) the plans (containing input and output parameters, actions and

interactions, and events) of MRs are modelled in a UML activity diagram; **MR Interaction Model:** As a result of (9) the interactions between MRs are derived and the exchanged objects (information carriers) are modelled in a UML sequence diagram; **MR Service Model:** As a result of (10) the signature of provided services (containing visibility, input and output parameters) of a MR are specified and modelled similar to a class in a UML class diagram again.

The results of activities (11), (14), (15) and (16), the **Autonomic Role Model**, the **AR Plan Model**, the **AR Interaction Model** and the **AR Service Model** are similar to the corresponding MR models. Further resulting models are: **AR Norm Model:** As a result of (12) and parallel to (11) the norms for ARs are specified according to desired self-* properties. Notice, a norm of an AR realizes a part of a certain self-* property of a system; **AR Analysis Model:** As a result of (13) the monitoring and analysis of events and data by an AR is modelled in a UML activity diagram as a premise for the right choice of a plan; **Interaction Protocol Model:** As a result of (17) the interaction protocols for the (direct/indirect) interactions between all types of roles are specified in a UML sequence diagram; **Autonomic Element Model:** As a result of (18) MRs and ARs are combined into AEs that are modelled similar to a class in a UML composition structure diagram again; **Autonomic Element Instance Model:** As a result of (19) the deployment of the AEs on to resources is defined similar to a UML deployment diagram. Note, activities (11)-(16) are logically separated and represent the way of self-* property development.

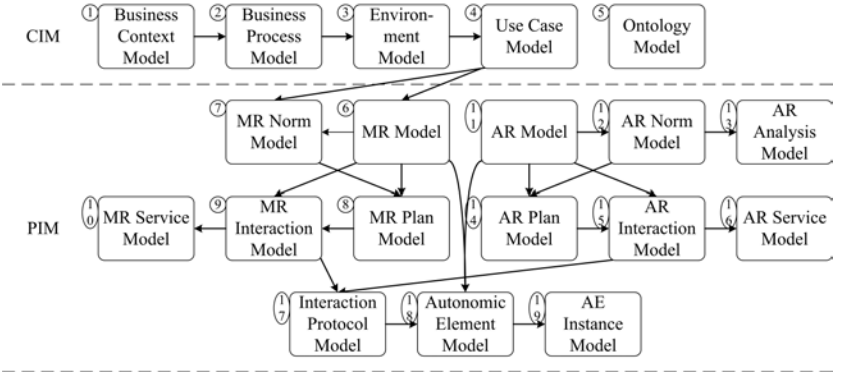

Fig. 4. MDA-based development process models for agent and OC systems

5 Case Study: Manufacturing Control

In order to evaluate the development process we have redesigned an existing MAS and added different self-* properties according to the proposed methodology to obtain an OCS. We will only illustrate activities (12), (13) and (14) by a short example to show in what way an existing MAS can be extended with self-optimization.

5.1 A MAS Production Planning System

Todays manufacturing industry is facing a major shift from a supplier's to a customer's market. Thus, the requirements on the manufacturing process itself increase permanently: Instant demand satisfaction, higher product variety and cost reducing are just some of them. Thus, Valckenaers *et al.* [29] developed a multi-agent coordination and control system based on stigmergy (coordination mechanism based on indirect communication, e.g. used by food foraging ants).

The system consists of three different types of agents: **Resource agents**, each assigned to a machine or switch in the manufacturing plant, **order agents**, each routing a product instance through the plant while reserving processing time on appropriate machines, and **product agents**, each containing the construction plan for a specific product. In addition, resource and order agents can make use of intelligent ant agents propagating and collecting information throughout the plant.

The coordination mechanism works as follows: resource agents assigned to machines permanently send ant agents opposite to the production line through the plant. These ants deposit the processing capabilities of their sending resource agent/machine as so called pheromones on every switch they cross. Order agents also create ant agents in a certain interval and send them down the production line. Based on the deposited pheromones on a switch, the ants decide to which machine they will travel next. When they arrive at this machine, they request an offer for a specific process step (e.g. duration, earliest start) and, as soon as they receive the offer, they continue travelling. If the end of the plant is reached, the ants return to their order agents and report their chosen route. An order agent decides which of its ants has found the best way and routes its product instance accordingly.

5.2 Adding Self-optimization

The existing system already copes with unforeseen machine breaks and short-termed changes. Nevertheless, the system can be improved by certain self-* properties, e.g. self-optimization.

Consider a single resource agent simultaneously responding to a multitude of offer requests of present ant agents, the response time can exceed in a way that

<<obligation>> **offer request response time optimization**
achieve: offer request response time in bounds
activation: offer request response time out of bounds
expiration: offer request response time in bounds

Fig. 5. Norm for self-optimization

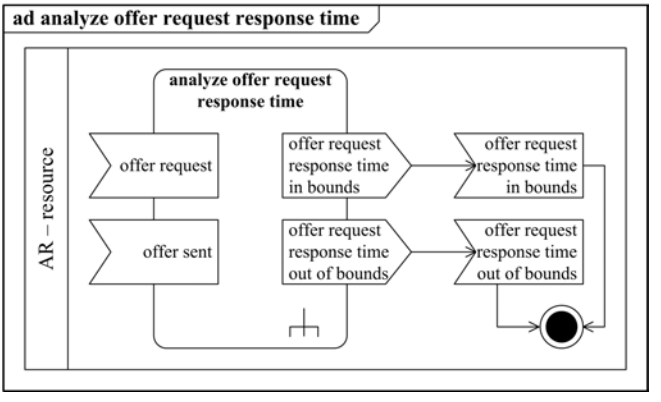

Fig. 6. Monitoring and analysis of events

the performance of the complete production system may slow down. In order to prevent this situation we added an autonomic role **AR-resource** to the managed role **MR-resource** which measures the response time and, if needed, informs the order agents to reduce their ant agent generation interval to minimize the amount of concurrent offer requests. For the measurement, the norm "offer request response time optimization" (see Fig. 5) has been specified for the AR-resource as result of (12). This "obligation" forces the AR to achieve the goal "offer request response time in bounds", is activated by the event "offer request response time out of bounds" and deactivated by the event homonymous to the goal.

To provide the AR-resource with the ability to analyse if this norm is activated or not, the monitoring and analysis (see Fig. 6) of specified events is modelled as

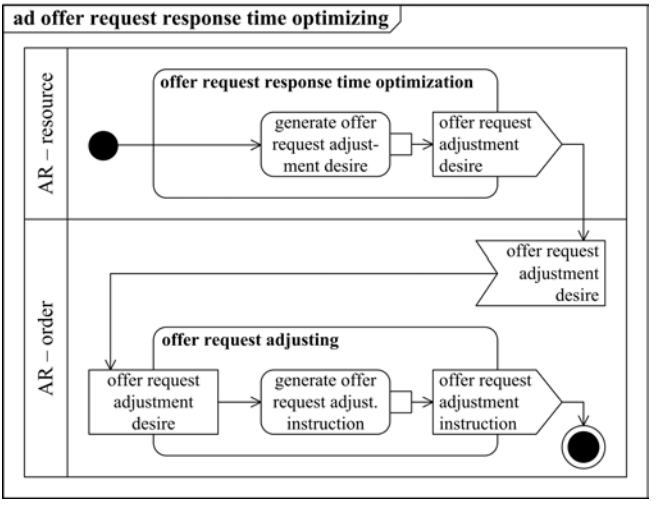

Fig. 7. Plans for Autonomic Roles

result of (13). The AR-resource listens to **offer request events** and **offer sent events** fired by the MR-resource. Within the analysis action, the response time is determined according to given rules (not modelled in this figure) and - depending on the result - a corresponding event is fired. By catching this event the AR-resource can determine whether or not the above norm is activated or deactivated.

If the analysis marks the norm as activated, the AR-resource has to choose a plan for informing the order agents to reduce their generation interval. Such a plan is modelled within an action (see Fig. 7) as result of (14). The AR-resource generates an **offer request adjustment desire** and sends it to every order agent or rather to every autonomic role **AR-order** (similar to a broadcast), possibly propagated by ant agents again. Such an AR in turn generates an **offer request adjustment instruction** for its controlled managed role **MR-order** which on its part slows down the generation interval (not modelled in this figure).

6 Conclusion, Open Issues and Outlook

As described in this paper, agent systems and OC systems have many conceptual commonalities that result in benefits for both AOSE and OC. On the one hand, open agent systems can be developed that exhibit OC properties; on the other, OC can make use of the experiences in AOSE and adopt existing concepts.

The open issues in this context for us are: Where are the borders between an autonomic element, an agent and a multi-agent system? How to deal with the emergent behaviour of the system such that no unintentional behaviour of the system occurs? How to define emergency strategies if the system is out of control, with regard to the emergent behaviour? Should we have an hierarchical composition, like grouping autonomic elements to autonomic communities, view these communities as autonomic elements and grouping them to autonomic communities, etc.? How to model self-* properties in the local as well as in the global sense and how does the local behaviours result in a global behaviour? How to integrate interaction (communication protocols) in such OC systems? What is the appropriate middleware/platform for OC systems (web services, grid computing middleware, agent platforms, ...)?

In this context our vision is to combine different but related technologies, like grid computing, semantic web, (semantic) web services and web service composition, P2P, business processes and OC with its self-* properties, since these technologies deal with similar aspects (service provisioning, service access, service and data distribution, service and resource work loading, processes in distributed environments) and use similar standards. Finally, the methodology has to be evaluated, validated and compared against other software engineering methodologies in the areas of AOSE and OC.

References

1. Jennings, N.R., Sycara, K., Wooldridge, M.J.: A Roadmap of Agent Research and Development. Autonomous Agents and Multi-Agent Systems, 1(1) (1998) 7–38
2. Wooldridge, M.: An Introduction to MultiAgent Systems. John Wiley & Sons, Chichester, England, 2002.

3. Huhns, M.N.: Multiagent Systems. Tutorial at the European Agent Systems Summer School (EASSS 99) (1999)
4. Organic Computing website: http://www.organic-computing.org
5. Horn, P.: Autonomic Computing: IBM's Perspective on the State of Information Technology. http://www.research.ibm.com/autonomic/manifesto/ autonomic_computing.pdf (2001)
6. Müller, J. P.: The design of intelligent agents. A layered approach. Lecture Notes of Artificial Intelligence, Volume 1177. Springer-Verlag (1996)
7. Huhns, M.N., Singh, M.P.: Agents and Multiagent Systems: Themes, Approaches, and Challenges. Readings in Agents, Morgan-Kaufmann (1998), 1–24
8. FIPA: http://www.fipa.org
9. Finin, T., Fritzson, R., McKay, D., McEntire, R.: KQML as an Agent Communication Language. Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94). ACM Press (1994) 456–463
10. IBM: An architectural blueprint for autonomic computing. http://www-03.ibm.com/autonomic/pdfs/ACBP2_2004-10-04.pdf (2004)
11. Di Marzo Serugendo, G., Gleizes, M.-P., Karageorgos, A.: Self-Organisation in Multi-Agent Systems. AgentLink News (16) (2004) 23–24
12. Bauer, B., Müller, J.P.: Methodologies and Modeling Languages. In: Luck M., Ashri R. D'Inverno M. (eds.): Agent-Based Software Development. Artech House Publishers, Boston, London (2004)
13. Iglesias, C.A., Garijo, M., Centeno-González, J.: A Survey of Agent-Oriented Methodologies. In Proceedings of Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL 98) (1998) 317–330
14. Kinny, D., Georgeff, M., Rao, A.: A Methodology and Modeling Technique for Systems of BDI Agents. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW 96), LNAI 1038, Springer (1996) 56–71
15. Glaser, N.: Contribution to Knowledge Modelling in a Multi-Agent Framework (the Co-MoMAS Approach). PhD thesis, L'Universtité Henri Poincaré, Nancy I, France (1996)
16. Iglesias, C.A., Garijo, M., Centeno-González, J., Velasco, J.R.: A methodological proposal for multiagent systems development extending CommonKADS. In Proceedings of 10th Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW 96), Banoe, Canada (1996)
17. Wooldridge, M., Jennings, N.R., Kinny, D.: The Gaia Methodology for Agent-Oriented Analysis and Design. Journal of Autonomous Agents and Multi-Agent Systems, 3 (3) (2000) 285–312
18. Juan, Th., Pearce, A., Sterling, L.: ROADMAP: Extending the Gaia Methodology for Complex Open Systems. In Proc. of the First Int. Joint Conf. on Autonomous Agents and Multiagent Aystems (AAMAS 02), ACM Press (2002) 3–10
19. Omicini, A.: SODA: Societies and Infrastructures in the Analysis and Design Of Agent-based Systems. In Proceedings of Agent Oriented Software Engineering (AOSE 00), LNCS 1957, Springer (2000) 185–193
20. Kinny, D., Georgeff, M: Modelling and Design of Multi-Agent Systems. Intelligent Agents III: Proceedings of Third International Workshop on Agent Theories, Architectures, and Languages (ATAL 96), LNAI 1193, Springer (1996)
21. Caire, G., Coulier, W., Garijo, F., Gomez, J., Pavon, J., Massonet, P., Leal, F., Chainho, P., Kearney, P., Stark, J., Evans, R.: Agent Oriented Analysis using MESSAGE/UML. In Proceedings of the Second International Workshop on Agent-Oriented Software Engineering II (AOSE 01), Springer (2002) 119–135

22. Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., Perini, A.: Tropos: An Agent-Oriented Software Development Methodology. Journal of Autonomous Agent and Multi-Agent Systems, 8 (3) (2004) 203–236
23. Padgham, L., Winikoff, M.: Developing Intelligent Agent Systems: A Practical Guide. John Wiley & Sons (2004)
24. Busetta, P., Rönnquist, R., Hodgson, A., Lucas, A.: JACK Intelligent Agents - Components for Intelligent Agents in Java. AgentLink News (2) (1999) 2–5
25. DeLoach, S.A., Wood, M.F., Sparkman, C.H.: Multiagent Systems Engineering. The International Journal of Software Engineering and Knowledge Engineering, 11 (3) (2001) 231–258
26. Cossentino, M., Potts, C.: A CASE tool supported methodology for the design of multi-agent systems. In Proceedings of the 2002 International Conference on Software Engineering Research and Practice (SERP'02), Las Vegas, USA (2002)
27. De Wolf, T., Holvoet, T.: Towards a Methodology for Engineering Self-Organising Emergent Systems. In Self-Organization and Autonomic Informatics (I), Volume 135 of Frontiers in Artificial Intelligence and Applications. H. Czap, R. Unland, C. Branki and H. Tianfield (editors), pp 18 - 34. Proceedings of the International Conference on Self-Organization and Adaptation of Multi-agent and Grid Systems (SOAS 2005), Glasgow, Scotland, UK.
28. Model Driven Architecture website: http://www.omg.org/mda
29. Valckenaers, P., Van Brussel, H., Kollingbaum, M., and Bochmann, O.: Multiagent coordination and control using stigmergy applied to manufacturing control. Multi-Agent Systems and Applications, LNAI 2086, Springer (2001) 317–334

An Agent-Oriented Model of a Dynamic Engineering Design Process

Vadim Ermolayev[3], Eyck Jentzsch[1], Oleg Karsayev[2], Natalya Keberle[3], Wolf-Ekkehard Matzke[1], Vladimir Samoylov[2], and Richard Sohnius[1]

[1] Cadence Design Systems, GmbH, Feldkirchen, Germany
{wolf, jentzsch, rsohnius}@cadence.com
[2] SPII RAS, Saint Petersburg, Russia
{ok, samovl}@iias.spb.su
[3] Zaporozhye National Univ., Zaporozhye, Ukraine
vadim@ermolayev.com, kenga@zsu.zp.ua

Abstract. One way to make engineering design effective and efficient is to make its processes flexible i.e. self-adjusting, self-configuring, and self-optimizing at run time. This paper presents the descriptive part of the Dynamic Engineering Design Process (DEDP) modelling framework developed in the PSI[1] project. The project aims to build a software tool to assist managers to analyse and enhance the productivity of the DEDPs through process simulations. The framework incorporates the models of teams and actors, tasks and activities as well as design artefacts as the major interrelated parts. DEDPs are modelled as weakly defined flows of tasks and atomic activities that may only "become apparent" at run time because of several presented dynamic factors. The processes are self-formed through the mechanisms of collaboration in the dynamic team of actors. These mechanisms are based on contracting negotiations. DEDP productivity is assessed by the Units of Welfare collected by the multi-agent system that models the design team. The models of the framework are formalized in the family of PSI ontologies.

1 Introduction

It is widely accepted that the processes of engineering design differ from manufacturing processes by the fact that they "… are frequently chaotic and non-linear, and have not been well served by project management or workflow tools" (cf. [1]). The primary reason is that the ability to design is one of the signatures of human intelligence that can hardly be framed by the rigid and static bounds of pre-defined business processes. Therefore, one of the promising ways to make engineering design effective and efficient is to manage its processes in a flexible manner i.e. make them self-adjusting, self-configuring and self-optimizing at run time. By doing so, we may enhance the degree of coherence among the interrelated activities and make them better coordinated and therefore more productive. Hence, the model of a DEDP

[1] Performance Simulation Initiative (PSI) is an R&D project of Cadence Design Systems GmbH.

M. Kolp et al. (Eds.): AOIS 2005, LNAI 3529, pp. 168 – 183, 2006.

should be at least capable of accounting for the many factors that make a DEDP "chaotic and non-linear" and, at most, to eliminate them as much as possible. Using software agents for optimizing DEDPs at runtime in dynamics is natural. Indeed, an agent by definition is capable of acting autonomously, pro-actively and rationally in pursuit of the desired state of affairs. Therefore, it may be used as the locus of self-configuration and self-optimization in a DEDP. Provided that we have built such a fine-grained, agent-oriented DEDP modelling framework, we may implement software tools allowing to assess a process and, ultimately, to optimize DEDPs in terms of engineering design productivity.

Improving DEDPs in terms of engineering design productivity is the focus of the PSI project. The project has prototyped a software tool that provides for the assessment of accomplished DEDPs and the prediction of the characteristics of planned DEDPs through their simulation. This simulation prototype has been implemented as a multi-agent system [2][2] which models: designers' teams working on projects by dynamically formed teams of software agents; DEDPs performed by these teams by tasks; the results of these processes by design artefacts. The knowledge about the performed processes is formalized and stored in the PSI testbed in terms of the PSI family of ontologies presented in this paper. Thus, we obtain an incremental collection of the actors' experience, which is further on re-used to make simulation results more reliable.

The paper is structured as follows: Section 2 discusses modelling requirements justifying the necessity of coping with the dynamic character of DEDPs. Section 3 outlines our approach to assessing the productivity of DEDPs. Section 4 presents the ontological model of a DEDP designed as a family of ontologies. Section 5 deals with the epistemological and usage aspects of PSI ontologies. It also briefly reports on the evaluation of the presented ontologies. Section 6 surveys the related work and analyses the contributions of the presented DEDP model. Section 7 concludes the work.

This is a substantially revised version of ourworkshop paper [3]. The revision has been undertaken to present the advancement we have achieved in the development of PSI ontologies and is based on their specification version 1.5 [4]. The negotiation part of this framework uses PSI Generic Negotiation Ontology [5].

2 Modelling Principles

A DEDP is understood as a weakly defined engineering design workflow. It aims to achieve its goal (the design artefact comprising a certain set of its representations) in an optimal way in the terms of result quality and gained productivity. It is therefore clear that the following entities are involved in the process: actors, who form design teams and collaboratively do the work in the flow; activities, which are the atomic parts of a workflow defined by the technology used in the house; tasks, which are the representations of hierarchical clusters of activities; and design artefacts, which are the results of engineering design activities. Hence, activities are defined by the design

[2] In this paper we omit the description of this important part of our research due to space limitations.

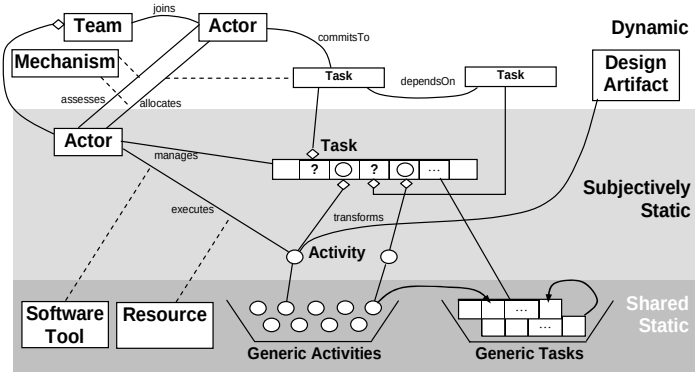

Fig. 1. Static and dynamic components of the modeling framework

technology and are well known before a DEDP starts. They form the "basket" of activities (Fig. 1), are uniformly understood and used by any actor and, therefore, may be considered generic. Another static shared "basket" is the one of generic tasks – please refer to the description below. Other elements may only "become apparent" at run time because:

- A generic activity may be executed only if it is assigned to an actor and is applied to a certain design artefact. Such an activity differs from a generic one by having particular associations to an actor and a design artefact. Task – Activity Ontology contains two separate concepts for a generic activity and an activity.
- Tasks are also distinguished as generic and as actually performed in the presented ontologies. A generic task is a shared static template defining a typical transformation of a design artefact from an initial set of representations to the target state. This transformation can be achieved by different combinations of generic activities. A task is subjectively dynamic because of its relationship to the specific actor who performs the task. This relationship is resolved as the result of the task assignment to an actor which happens at run time, when the DEDP is performed. Task – Activity Ontology contains two separate concepts for a generic task and a task.
- A task is the model of the emerging hierarchical structure of a DEDP or the part of a DEDP. It may contain tasks and activities as its integral parts. The main purpose of a task is to arrange the assignment of its sub-tasks and activities. This arrangement is done by the actor (having the function of the task manager) who performs the task. The assigned sub-tasks may be consequently arranged in the similar manner by their task managers until the "leaves" of the hierarchical structure (the activities) are reached. These activities are assigned to and executed by the actors. By formalizing the above, we define the model of the cascade decomposition of tasks and, ultimately, a DEDP at run time.
- The number of activity loops is not defined in advance. It depends on the quality checks at intermediate steps. Changing the number of activity loops may cause changes in activity duration. In turn, it may cause delays of the dependent tasks and activities with associated penalties for deadline violation, for example.

- The duration of activity execution is not defined in advance. Different actors are able to spend different capacities to execute the same activity at a certain time. Actors may perform the same activity with different efficiencies (productivities – Section 3). An activity may remain idle while waiting until the pre-conditions have been triggered. Idle state duration can't be computed in advance because the preconditions may be formed by other activities executed by other actors.
- An actor is chosen by the task manager when s/he decides to assign the activity. In the PSI framework contracting negotiations are the means of optimally choosing the actors to perform the tasks. For the planning phase it means "optimally" from the point of view of the project manager. The DEDP model incorporates the actor model and the means to arrange actors' collaboration through peer assessment and negotiations.

The abovementioned factors provide certain degrees of freedom[3] in DEDP planning, re-planning, scheduling, re-scheduling and execution. In PSI a DEDP is never rigidly planned before it starts. The decisions of how to continue its execution are made each time it reaches a certain state in the state space. These decisions are made by the design team members who manage the tasks that continue the process. According to the aforementioned properties of a DEDP, different paths through the state space may be more productive or less.

As shown in Fig. 1, a DEDP has components that differ along the dimensions of their variability. The first dimension is the dynamic character ranging from static, i.e. pre-defined for all possible DEDPs, to dynamic, i.e. subjected to changes in a DEDP. Another dimension is the sphere of visibility or commitment. This dimension ranges from shared, i.e. having the same meaning and instances for all DEDP participants, to subjective, i.e. having specific instances for different actors (though in the terms of a common ontology). Static shared DEDP components are generic activities, associated software tools and resources. The model of a DEDP assumes that the processes are assembled (ultimately) of atomic activities, which are the pieces of the design technology used by the company. The technology normally provided by a design support unit often suggests the usage of a specific software tool to perform an activity. The execution of a given activity consumes certain resource instances in given quantities. The model of a design process is based on the following assumptions: a DEDP is initiated by an external influence providing a goal to a certain actor. This goal is subjectively transformed to a task according to the knowledge of this actor. The actor uses his subjective knowledge about sub-tasks and activities to decompose a task. The actor may decide to perform a sub-task or to execute an activity of a decomposed task themselves or to hire another actor for a price in Units of Welfare (Section 3) using the available collaboration mechanism (contract net negotiations in PSI). In the latter case, the sub-task becomes the goal of another peer-actor who commits to performing the corresponding task by striking the contract deal. Hence, the appearance of actor-task combinations in a DEDP is subjectively dynamic. The mechanism of incorporating new actors to the process and the model of the design team are subjectively dynamic as well, since they depend on the decisions and choices

[3] It should be noted here that this freedom implies more complications in planning, scheduling and the necessity to deal with a finer grained DEDP model.

taken at run time by the actors the state of which can change in the process. The rules of encounter of the mentioned mechanism are shared static and provide the horizontal laws for the system [6, 7].

A design artefact is a subjectively dynamic outcome of the process since it is formed out by a subjectively dynamic collaborative team of actors. However, the proposed layering allows reaching this effect through applying shared static atomic activities, though in subjectively dynamic combinations. For an activity, a design artefact is both the material input and the result of its execution.

The actors who perform a task and initiate collaboration are Task Managers. Their rational goal with respect to the performed task is to choose the next step on the process path as productively as possible. Of course, for that, an actor needs a sort of productivity assessment model.

3 Assessing Productivity by the Earned Units of Welfare

Productivity by its very nature is one of the most important economic metrics and is defined by the ratio of the produced output (value) to the consumed input (value). As such, it is an integral characteristic of any transformation process, e.g. a DEDP. This neo-classical definition of productivity imposes rigid requirements on the process under consideration. The homogeneity of inputs and outputs is the most severe one with respect to engineering design. Known productivity measurement methodologies in engineering design ground themselves on the assessment of design complexity characteristics in the creation of homogeneous input- and output-measures. They do it by applying heuristic weights to compared parameters (e.g. the normalized transistor count[4] in Semiconductor and Electronic Systems (SES) design, FP, KSLOC counts[5] in software design etc.). But the fundamental problem of this approach is that the complexity characteristics need to be invariant both to the type of a process and to the transformed design artefact. If those characteristics are not invariant, measurement scales tend to lack well-defined units. Consequently, the properties of the measurement scale, the labelling of the units and the interpretation of the values derived are of very limited practical use. Furthermore, in non-deterministic environments such measures are not very reliable. It is therefore important to build a measure that addresses the homogeneity requirement with respect to inputs and outputs and that is invariant to the dynamic characteristics of a process (Section 2). Such a measure may be based on the integral process success indicators like, for example, the ratio of the Earned Value to the Planned Value or to the Actual Cost at a Sign-off Stage of the process. This implies that productivity of a DEDP may be assessed by the value produced and accumulated by designers in a team. The more value produced by a designer, the more relatively productive s/he is. It is also true in the longer term if several DEDPs are taken into consideration. Hence, more productive designers are characterized by the higher volume of accumulated Units of Welfare (UoW). It is assumed in PSI that designers receive incentive which is adequate to their produced value. The characteristic of UoW assets is invariant to all

[4] Measuring IC and ASIC Design Productivity. White Paper. Numetrics Management Systems, 5201 Great America Parkway, Suite 320 Santa Clara, CA 95054, 2000.

[5] FP stands for Functional Point, KSLOC – for kilo lines of source code.

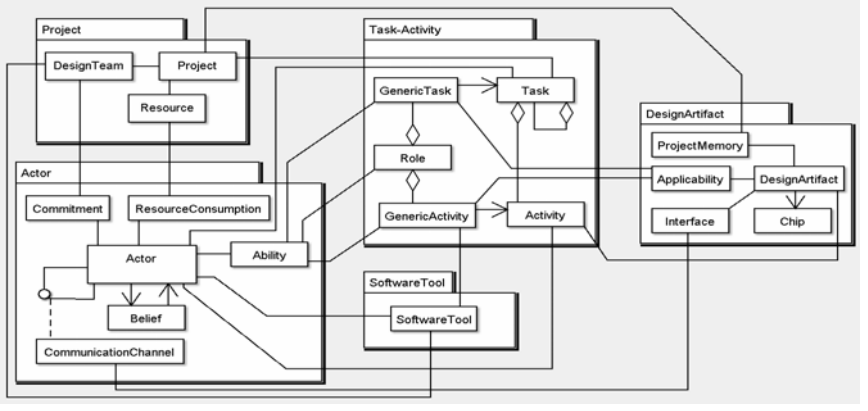

Fig. 2. High-level structure of the family of PSI ontologies

aforementioned dynamic features of an engineering design process. It is a normalized scalar measure, which, by its semantics, is similar to the notion of Utility that is used in Distributed Rational Decision Making. UoW earning and spending mechanisms in PSI are based on contracting deals struck in several types of negotiations [7].

4 Overview of PSI Ontologies

If we intend to model an arbitrary process of doing something (for example, a design process), the basic building blocks for such a model would be: a goal – the state of affairs to be reached; an action; an object to apply actions to; a subject who applies actions to objects; an instrument to be used by a subject to execute actions; and an environment in which the process occurs. The structure of the PSI ontologies family reflects this approach (Fig. 2). It comprises five tightly linked major ontologies which in UML representation are grouped in separate packages: the Actor Ontology (a subject), the Project Ontology (an environment), the Task-Activity Ontology (an action), the Software Tool Ontology (an instrument) and the Design Artefact Ontology (a goal and an object). The classes shown within the packages in Fig. 2 identify the major concepts of the respective ontologies.

This grouping of course reflects the principles of the modelling approach (Section 2). Indeed, the outline given in Fig. 1 and the high-level picture of the family of PSI ontologies have many features in common.

4.1 Actors, Beliefs, Collaboration, Design Teams

Actors are the models of designers who form Design Teams to perform design Projects (Fig. 3[6]). As the members of a Design Team they have certain Commitments with respect to the Design Team and to the Project under performance. Actors perform (i.e. manage) Tasks and execute Activities, which transform certain Design

[6] Yellow UML packages in Fig. 3, 4, 5 represent the ontologies that are external to the described one.

Artefacts. Actors use Software Tools to execute Activities and have certain Attitudes to these Software Tools. Actors consume Resources to execute Activities. One of the important aspects of the Actor model is the representation of Actor's Ability to perform Tasks, execute Activities and perform organizational Roles (like Front-End Designer, Back-End Designer etc.). With respect to an abstract Generic Activity, a Generic Task and a Role, we are also interested if an Actor is able (and to what extent) to execute the atomic action, or to perform a task or a role. The association concepts of an AbilityWrtActivity, an AbilityWrtTask and an Ability provide the means for the answers. The Actor ontology also describes Communication Channels that emerge among different Actors who collaboratively work on different parts of the specific Design Artefact. The reason for such collaboration is the peculiarity of the structure of the Design Artefact. Collaboration occurs among the Actors who work on parts of the Design Artefact having common Interfaces. Another kind of collaboration among the members of a Design Team is their coordinated performance of Tasks and execution of Activities. This teamwork is arranged via contracting negotiations. An actor playing the Role of a task manager intends to outsource a Task to one of his or her peers. The following two aspects constrain the set of peer-Actors to the sub-set of the believed performers: a task manager believes that the believed performers are (i) capable to perform the Task and (ii) credible enough to trust the performance of the Task to them. These Beliefs are modelled by the Belief concept of the Actor ontology. Beliefs are the important part of the negotiation mechanisms and are related to the concepts of Negotiation Outcomes and Negotiation Strategies of the Generic Negotiation ontology [5]. The Beliefs of the negotiation participants are updated according to the outcome of the negotiation. Beliefs are used by Actors to adjust their negotiation strategies providing the assessments of the peers' capabilities and credibility [6]. An Actor becomes the contractor for the negotiated generic task or generic activity in result of such a negotiation. He is the only one of believed performers who receives the negotiated Task (Activity) and commits to performing (executing) it as a member of the Design Team. A Design Team is the bridge providing the relationship of a DEDP to the Project, which is implemented performing this DEDP.

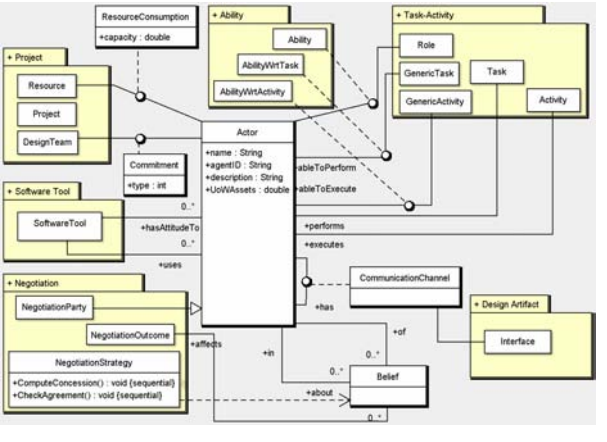

Fig. 3. Outline of the PSI **Actor** ontology

4.2 A Project as the Environment of a DEDP

The Project ontology describes the environment of a DEDP: an organizational structure around its performance. This ontology at the high level resembles the traditional project planning perspective, which states that a process is performed by a team (of Actors) and has a collection of Resources associated with it. A Process is viewed as a sequence of transformations of the target Design Artefact. These transformations may be viewed as the transitions between the States of a Design Artefact. The objective of each transformation is to develop the increment of a Design Artefact in a certain Representation. The States in this transformation process are therefore characterized by the addition of the certain Design Artefact Representations to the Design Artefact under transformation. In the process of this transformation, a Design Artefact receives its incremental "slices" at particular States. One such "slice" bijectively corresponds to one instance of a Design Artefact Representation. Representations are booked to the Project Memory. Please see also Section 4.4.

4.3 Tasks, Activities, Co-execution, and Dependencies

The purpose of the Task-Activity ontology (Fig. 4) is to provide the descriptive framework for modelling the emerging dynamic hierarchical structure of a design process.

As outlined in the description of the modelling approach, only Activities are executed. An Activity is understood as the atomic purposeful action thatis applied to a certain Design Artefact and results in its transformation from one State to another State adding a Representation "slice" to it. For example, the *RTL*[7]*-Design* Activity uses a Design Artefact in the *specification* representation and transforms this Design Artefact by adding the *RTL* representation. However, an Activity applied to different Design Artefacts results in different outcomes. Indeed, the *RTL-Design* Activity applied to *FB1* or to *FB2* – instances of a Design Artefact – will have *FB1* in the target representation of *RTL* or *FB2* in the target representation of *RTL* respectively as its outcomes. On the other hand, the same activity, even applied to the same *FB1* but executed by different Actors, may require different efforts to be spent to achieve its outcome. That is why the ontology introduces the concept of an Activity assuming its particular association with an Actor and a Design Artefact.

A Generic Activity is the more abstract concept that denotes or "shapes-out", as the relationship name suggests, a purposeful, atomic action. This action is actually the transformation that is configured by the State Pattern of an (intended) Design Artefact. This State Pattern is the template that configures the inputs and outputs of the related Generic Activity. These abstract inputs and outputs may receive physical materialization as the particular instances of a Design Artefact (or its sub-concepts) only when the Activity corresponding to the configured Generic Activity is executed. The action specified by the Generic Activity may be performed using a Software Tool – either one specific, or several alternative ones. One more important aspect captured by the context of a Generic Activity is the relationship to co-executed activities. The model provided by the ontology allows specifying that a pair of activities may be

[7] RTL stands for Register Transfer Level.

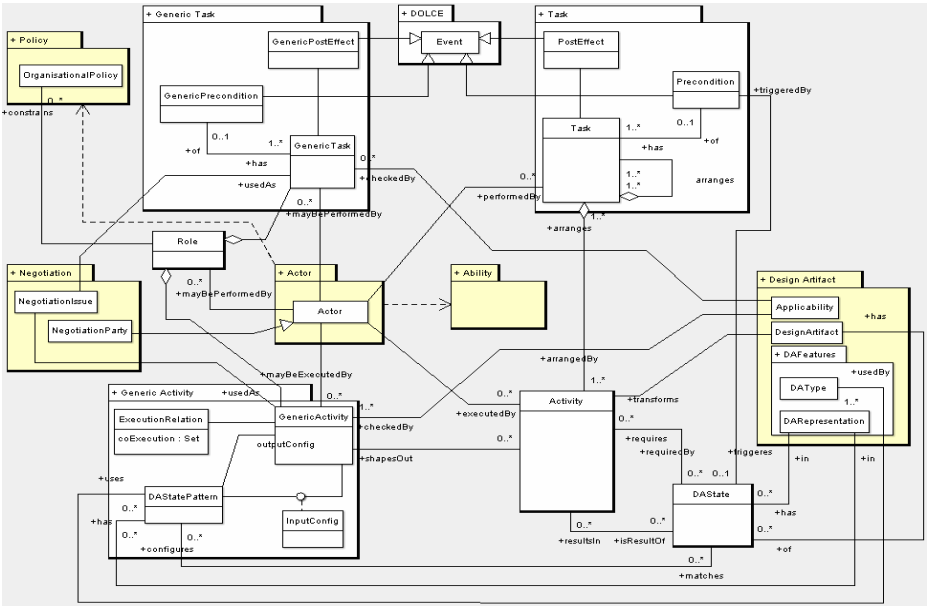

Fig. 4. Outline of the PSI **Task-Activity** ontology

executed in an arbitrary sequence or should be executed in parallel. For example, the *RTL Debug* and *RTL Verification* activities are to be executed in parallel because executing only one of them makes no sense according to the design technology. This part of the model is used in resolving the decomposition of Generic Tasks to Generic Activities at the Work Breakdown Structure generation phase of project planning.

The concept of an Activity refines the concept of a Generic Activity by providing the new knowledge about the assigned Actor and the transformed Design Artefact through its relationships. A Generic Activity in contrast to an Activity is the abstract specification of an atomic action. These atomic actions are executed by Actors as Activities aimed to transform Design Artefacts in source representations into Design Artefacts in target representations.

The concepts of a Generic Task and a Task have the similar relationship to each other. A Task is performed (managed) by the certain assigned Actor. However, the semantics of this pair of concepts is different from the ones describing activities. A Task is the concept that (i) describes the dynamic hierarchical nature of a design process; (ii) may contain sub-tasks of lower granularity as its integral parts; (iii) may wrap a single or a set of Activities under the umbrella of the single Actor who is the task manager. A Task refines the concept of an (abstract) Generic Task by being related to an Activity and an Actor. A Generic Task is the abstract template of a composite action (in difference to the atomic abstract action modelled by a Generic Activity). Tasks may also be viewed as the abstract descriptions of capabilities used to form Roles.

The presented Task-Activity model handles the dependencies among Tasks and among Activities assuming that these dependencies are strong [3]. These dependencies are resolved by using the knowledge about the initial and target states of the

associated Design Artefacts (Section 4.4). A representation of the Design Artefact reached in a particular state (by the execution of certain Activities) is required by the dependent Activity. This state also triggers the Precondition of the dependent Task if the dependencies among Tasks are analysed. A Task results in certain Post-Effects. A Post-Effect is the event of reaching the particular state by the processed Design Artefact (DA). A similar model is used for capturing the dependencies among Generic tasks or Generic Activities. The difference is in the semantics of the corresponding concepts. For example, a DAStatePattern does not denote the state(s) of a Design Artefact, but is rather the template used for the configuration of the intended inputs and outputs of a Generic Activity.

As described in Section 4.1, Generic Tasks and Generic Activities are used as Negotiation Issues in the negotiations on the assignment of Tasks and Activities to certain Actors.

4.4 Design Artefacts

The central concept of the Design Artefact ontology (Fig. 5) is Design Artefact – the goal and the object of a design process. At a high level, the ontology focusses on the following aspects of this model: (i) a Design Artefact as the object of the transformation process is related to the executed actions i.e. to the concept of an Activity; (ii) a Design Artefact (more exactly the Functional Block of the topmost level) as the goal of the process is materialized in a Chip[8] – the terminal state of affairs to be achieved; (iii) a Design Artefact as a complex structure comprising different integral parts in different representations may induce collaboration of different Actors by indicating common Interfaces of its integral parts; (iv) the trace of Design Artefact transformations and the related states are recorded into the Project Memory. A Project Memory, therefore, provides a link of a Design Artefact transformation trace to the design process environment.

From the point of view of domain grounding, the ontology specifies that a Design Artefact comprises the hierarchy of Functional Blocks as the structural elements of designed functionality. Functional Blocks are generally viewed as “grey boxes” with functional subdivision defined by the taxonomy of Design Artefact Types. The top-level examples of these types are: *digital*, *analog*, *mixed-signal*. The taxonomy of types also configures the Applicability of Generic Tasks and Generic Activities to a Design Artefact. The reason is that the technology and, therefore, the subsets of applicable tasks and activities are different for different types of design artefacts.

The instances of a Functional Block are complemented by the instances of the other subclasses of a Design Artefact – Documentations, TestBenches and Verification Runsets – the means to document, test and verify designs according to the provided engineering design technology.

[8] Design Artefacts may not be materialized in a Chip in some design processes. For example, a process having the goal to design Soft IP will have a different Design Artefact Representation (GDS2 or NetList) as the terminal one. Such Soft IPs are often released in Libraries for further re-use in different design processes and projects. However, Design Artefacts in Semiconductor and Electronic Systems domain are designed to be sooner (in the current process) or later (in another process) materialized in a Chip.

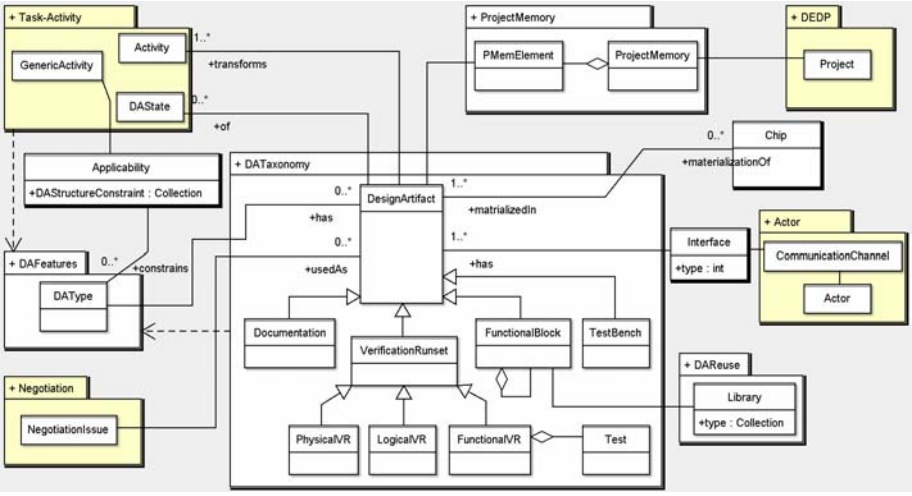

Fig. 5. Outline of the PSI **Design Artefact** ontology

Design Artefacts are used as Negotiation Issues in a DEDP. The typical cases are: (i) an Actor looks for a Soft IP to be re-used in his current design and negotiates the terms of usage with the owners of this IP; (ii) a Design Artefact is one of the issues in the multi-issue negotiation on the assignment of a Task to an Actor.

4.5 Software Tools

The Software Tool ontology focusses on the description of the two aspects of an instrument used by Actors to execute Activities. The first aspect is the instrument itself. A Software Tool is used by an Actor to execute an Activity. The second aspect is the usefulness of a Software Tool. Different Actors while using the same Software Tool may be somewhat productive. Therefore, Actors may have different Attitudes to a certain Software Tool. Though these subjective attitudes are important they, if analysed separately, do not provide a reliable judgment. Therefore the average impression provided by a Design Team may be more useful for Evaluation purposes. A Software Tool has these relationships to the concepts of an Actor and a Design Team.

5 PSI Ontologies: Epistemology, Usage and Evaluation

The descriptive part of the DEDP modelling framework has been initially designed as a family of ontologies and coded in a set of UML class diagrams. Further formalization and implementation work has been performed in the way aligned with scenarios of ontology usage identified by Uschold and Jasper [10]. PSI ontologies are used [2] for authoring DEDP logs recorded to the PSI testbed, for specifying the design of the DEDP-PMS simulator software and as shared ontologies for agent communication at run-time. Ontology usage aspects influenced the choice of the

formal languages for coding the ontologies. The ontologies were coded in OWL-DL[9]. This language was chosen because it is one of the de-facto ontology specification standards. The second reason for choosing OWL-DL was that its expressive power is similar to that of the internal mental model specification language (MMSL) of MASDK [11], which has been used for specifying the design and prototyping of the PSI prototype – DEDP-PMS. From an epistemological viewpoint, the transformation of the PSI ontologies to OWL-DL representation required the change of UML associations to constructs with binary relationships with restrictions. This transformation has been performed manually with the help of the Protégé 3.0[10] ontology editor as described in [4].

DEDP-PMS[11] has been implemented to evaluate the modelling framework, to experiment with several planning and scheduling algorithms, and to assess the feasibility of building a software tool for DEDP optimization using their productivity assessment. Two rounds of evaluation experiments have been performed. The first round has been done over the two simplified test cases (the DEDPs for the Digital and the Analog DAs) and used version 1.0 of the PSI family of ontologies. The second round used version 1.4 of the PSI family of ontologies and has been applied to a real world case study [8]. In the first round of evaluation experiments [2], the simulator has been used in two application modes: playback and predictive simulation. In playback mode, the simulation is used to assess the performance of DEDPs that have been accomplished in the past. The purpose of the predictive simulation is to support project managers in planning and dynamic re-planning of running design projects in the cases the occurrence of several kinds of events that are out of their control: late changes to the design objective, sudden unavailability of team members, changes in the workload of the designers according to the influence of other independent projects etc. In the first round of evaluation experiments, only the availability of the actors has been altered by random "screwing" of the corresponding simulation parameters. The second round has been focussed on the evaluation of the dynamic planning capabilities. The goal of the experiments was to compare the Work Breakdown Structure automatically generated by the DEDP-PMS with the one created manually by the project manager. The details of these experiments are described in [8].

Evaluation experiments with the available DEDP records stored to the PSI testbed demonstrated that the simulator develops DEDP plans very closely to what happened in reality i.e. the plans developed by human project managers. Observed fluctuations were caused by the changes in the parameters of the availability of team members in the course of the simulation experiments by "screwing" their available capacities. This fact confirms the adequacy of the developed framework to the industrial requirements in Semiconductor and Electronic Systems Domain.

6 Related Work and Discussion

The projects that pioneered R&D in agent-based engineering design process modelling, support and automation appeared about a decade ago e.g. [12, 13, 14].

[9] OWL Web Ontology Language Reference. http://www.w3.org/TR/owl-ref/

[10] Protégé ontology editor and knowledge acquisition system http://protege.stanford.edu/

[11] DEDP-PMS has been presented at the 17th European Conference on Artificial Intelligence, ECAI 2006 [27].

Some projects of the "second wave" [15, 16] helped to specify the focus of PSI in automating the near-optimal arrangement of DEDPs in terms of their productivity. In contrast to e.g. [26], the objective of PSI is not to automate the design process itself but to automate the arrangement of its activities in the most productive way. In PSI, the activities resulting in the elaboration of design artefacts are performed manually by human designers.

The DEDP modelling framework in its part of organizational and actor-related knowledge representation is based on the frameworks [17, 18, 5, 9]. PSI contribution in this part is the incorporation of roles and actors, teams of actors, negotiation context in one coherent family of ontologies and the binding of these ontologies to the engineering design domain by incorporating Design Artefact and Software Tool ontologies. The main contribution of the PSI family of ontologies is the model of a dynamic team of designers that is formed through contracting negotiations and performs dynamically orchestrated processes. Hence, DEDPs in PSI are understood as socially performed processes in the sense close to [19]. For example, the notion of a Role of PSI Actor ontology is semantically close to that of the normative multi-agent framework.

In the part of process modelling, PSI bases its approach on [20, 6, 21]. In the family of PSI ontologies, engineering design processes are modelled as tasks composed of sub-tasks and atomic activities. Similarly to [22], subtasks and activities may have strong dependencies. However, in PSI, the knowledge about these dependencies is presented in a different way. The means for that are DA State Patterns and Execution Relations. DA State Patterns are the patterns of DA States. These are the concepts to configure the inputs, the outputs and the dependencies caused by the usage of the outputs as inputs. By that, the dependencies among activities are also aligned with the corresponding DA States. Execution Relations are used to represent concurrency among the activities in a pair. This concurrency may be caused by the specifics of the DA structure. Similarly to [21], tasks have pre-conditions and post-effects. However, the Task-Activity ontology constrains the semantics of pre-conditions and post-effects by making them sub-classes of an event concept. Material inputs and outputs [3] are modelled in frame of DA States.

Examples of theoretical frameworks for solving planning tasks are Decision Theoretic Planning (DTP) [23] and Hierarchical Task Networks (HTN) [24]. The PSI framework is built upon the conceptual denotation of the planning task shared by the previously mentioned frameworks. Planning is understood as the process of cascade decomposition of the goal, transformation of the sub-goals to Generic Tasks, Generic Activities and committing Actors to Tasks and Activities. However, the PSI framework extends the capabilities of the classical AI approaches to planning by accounting for the dynamic character of the process and by the capability to collaborative distributed planning through negotiation mechanisms. The latter feature also distinguishes our descriptive framework from the plan-task ontology of KMI [25]. Moreover, the family of PSI ontologies provides conceptual means for dynamic re-scheduling based on the concepts of self-beliefs and beliefs.

7 Conclusions

This paper has presented the descriptive part of the DEDP modelling framework developed in the PSI project. The project is aimed to build a software tool assisting in analysis and optimization of DEDPs' productivity through agent-based simulations. The framework incorporates the models of projects, teams and actors, tasks and activities, design artefacts, and software tools as the major interrelated parts. DEDPs are modelled as weakly defined flows of tasks and atomic activities. These flows are transformation processes. They transform design artefacts passing through the sequence of their states. DEDPs may "become apparent" only at run time because of several factors that are beyond the control of the design team members. The processes are self-formed through the mechanisms of collaboration in the dynamic team of actors. These mechanisms are based on several types of negotiations. DEDP productivity is assessed by the Units of Welfare collected by the multi-agent system that models the design team. The models of the framework are formalized in the family of PSI ontologies. These ontologies are used in the implemented simulator software prototype. Evaluation experiments have been performed using the PSI testbed [2, 8]. These experiments showed that DEDP planning performed using the DEDP-PMS software prototype reflects reality. Generated DEDP plans are very close to that developed by human project managers.

References

1. Neal, D., Smith, H. and Butler, D.: The evolution of business processes from description to data to smart executable code – is this the future of systems integration and collaborative commerce? *Research Services Journal*, 3 (2001) 39–49
2. Gorodetsky, V., Ermolayev, V., Matzke, W.-E., Jentzsch, E., Karsayev, O., Keberle, N. and Samoylov, V.: Agent-Based Framework for Simulation and Support of Dynamic Engineering Design Processes in PSI. In: Pechouchek, M., Petta, P., Varga, L. Z. (eds.): Proc. 4th Int. Central and Eastern European Conf. on Multi-Agent Systems (CEEMAS'05), Sept. 15-17, Budapest, Hungary, LNAI Vol. 3690. Springer-Verlag, Berlin Heidelberg New York (2005) 511–520
3. Ermolayev, V., Jentzsch, E., Karsayev, O., Keberle, N., Matzke, W.-E. and Samoylov, V.: Modeling Dynamic Engineering Design Processes in PSI. In: Akoka, J. *et al.* (eds.): ER Workshops 2005, Proc. 7th Int. Bi-Conf. Workshop on Agent-Oriented Information Systems (AOIS-2005), Oct. 24-28, Klagenfurt, Austria, LNCS Vol. 3770. Springer-Verlag, Berlin Heidelberg New York (2005) 119–130
4. Ermolayev, V., Jentzsch, E., Keberle, N., Samoylov, V. and Sohnius, R.: The Family of PSI Ontologies V.1.5. Reference Specification. Technical Report PSI-ONTO-TR-2006-2, 14.04.2006, Cadence Design Systems, GmbH (2006) 56 pp.
5. Ermolayev, V. and Keberle, N.: A Generic Ontology of Rational Negotiation. In: Karagiannis, D., Mayr, H.C. (eds.): Information Systems Technology and its Applications. 5th Int. Conf. ISTA'2006, May 30-31, Klagenfurt, Austria, LNI Vol. 84. Gesellschaft für Informatik, Bonn (2006) 51–66
6. Ermolayev, V., Keberle, N., Kononenko, O., Plaksin, S. and Terziyan, V.: Towards a framework for agent-enabled semantic web service composition. *Int. J. of Web Services Research*, 1(3) (2004) 63–87

7. Ermolayev, V., Jentzsch, E., Matzke, W.-E., Schmidt, J., Schroeder, G., Weber, S. and Werner, J.: Agent-Based Dynamic Engineering Design Process Modeling Framework. Technical Report. Cadence Design Systems, GmbH (2004) 29 pp
8. Sohnius, R., Ermolayev, V., Jentzsch, E., Keberle, N., Matzke, W.-E. and Samoylov, V.: Managing Concurrent Engineering Design Processes and Associated Knowledge. To appear in: Proc 13th ISPE Int. Conf. on Concurrent Engineering: Research and Applications, Sept. 18-22, Les Antibes, France, available at: http://ermolayev.com/eva_personal/evapubs.htm
9. Ermolayev, V. Keberle, N. and Tolok, V.: OIL Ontologies for Collaborative Task Performance in Coalitions of Self-Interested Actors. In: Arisawa, H. et al (eds.): Conceptual Modeling for New Information Systems Technologies. ER 2001 Workshops, Nov. 27-30, Yokohama, Japan. LNCS Vol. 2465. Springer-Verlag, Berlin Heidelberg New York (2002) 390–402
10. Uschold, M. and Jasper, R.: A Framework for Understanding and Classifying Ontology Applications. In: 12th Workshop on Knowledge Acquisition, Modeling and Management (KAW'99), Oct. 16-21, Banff, Alberta, CA (1999)
11. Gorodetski, V., Karsaev, O., Samoilov, V., Konushy, V., Mankov, E. and Malyshev, A.: Multi Agent System Development Kit: MAS software tool implementing GAIA Methodology. In: Shi, Z., He, Q. (eds.): Int. Conf. on Intelligent Information Processing (IIP'2004), Beijing, China Springer Press (2004) 69–78
12. Cutkosky, M.R., Englemore, R.S., Fikes, R.E. Genesereth, M.R. Gruber, T.R., Mark, W.S., Tenenbaum, J.M. and Weber, J.C.: PACT: An Experiment in Integrating Concurrent Engineering Systems. *IEEE Computer*, 26(1) (1993) 28–38
13. Darr, T. P. and Birmingham, W. P.: An Attribute-Space Representation and Algorithm for Concurrent Engineering. CSE-TR-221-94, University of Michigan, Department of Electrical Engineering and Computer Science, Ann Arbor, Michigan (1994)
14. Balasubramanian, S. and Norrie, D. H.: A Multi-Agent Intelligent Design System Integrating Manufacturing and Shop-Floor Control. In: Proc. 1st Int. Conf. on Multi-Agent Systems (ICMAS'95), San Francisco. The MIT Press, Cambridge (1995) 3–9
15. Parunak, H.V.D., Ward, A.C., Fleischer, M. and Sauter, J. A.: The RAPPID Project: Symbiosis between Industrial Requirements and MAS Research. *Autonomous Agents and Multi-Agent Systems*, 2(2) (1999) 111–140
16. Danesh, M. R. and Jin, Y.: An Agent-Based Decision Network for Concurrent Engineering Design. *CERA*, 9(1) (2001) 37–47
17. Fox, M.C. and Gruninger, M.: Enterprise Modelling. *AI Magazine*, 19(3): 109 – 121, 1998
18. Uschold, M. King, M., Moralee, S. and Zorgios, Y.: The Enterprise Ontology. *The Knowledge Engineering Review*, 13(1) (1998) 31–89
19. Boella, G. and van der Torre, L.: An Agent Oriented Ontology of Social Reality. In: Varzi, A., Vieu, L. (eds.): Proc. 3rd Int. Conf on Formal Ontology in Information Systems (FOIS'04), Torino, Italy, Nov. 3-6. IOS Press (2004) 199–209
20. Buhler, P. and Vidal, J.M.: Enacting BPEL4WS Specified Workflows with Multiagent Systems. In: Proc. of AAMAS'04 Workshop on Web Services and Agent-Based Engineering (WSABE) (2004)
21. Fensel, D. and Bussler, C.: The Web Service Modeling Framework WSMF. *Electronic Commerce Research and Applications*, 1(2) (2002) 113–137
22. Nagendra Prasad, M. V. and Lesser, V. R.: Learning Situation-Specific Coordination in Cooperative Multi-Agent Systems. *Autonomous Agents and Multi-Agent Systems*, 2(2) (1999) 173–207
23. Blythe, J.: Decision-Theoretic Planning. *AI Magazine*, 20(2) (1999) 37–54

24. Erol, K., Hendler, J. and Nau, D. S.: Semantics for Hierarchical Task-Network Planning. Technical report CS-TR-3239, University of Maryland at College Park (1994)
25. Rajpathak, D. and Motta, E.: An Ontological Formalization of the Planning Task. In: Varzi, A., Vieu, L. (eds.): Proc. 3rd Int. Conf on Formal Ontology in Information Systems (FOIS'04), Torino, Italy, Nov. 3-6. IOS Press (2004) 305–316
26. Capera, D., Picard, G. and Gleizes, M.-P.: Applying ADELFE Methodology to a Mechanism Design Problem. In: Proc. 3rd Int. Joint Conf. AAMAS'04, Aug. 19-23, New York, NY, USA. IEEE Computer Society (2004) 1508–1509
27. Samoylov, V., Gorodetsky, V., Ermolayev, V., Jentzsch, E., Karsayev, O., Keberle, N., Matzke, W.-E. and Sohnius, R.: Agent-based Prototype of the Dynamic Engineering Design Process Performance Management System (DEDP-PMS). Presentation at ECAI 2006, Aug.28 - Sept.1, Riva del Garda, Italy, 2006. Abstract is available from: http://ermolayev.com/eva_personal/evapubs.htm

Formalizing Agent-Oriented Enterprise Models

Ivan Jureta[1], Stéphane Faulkner[1], and Manuel Kolp[2]

[1] Information Management Research Unit, University of Namur,
8 Rempart de la vierge, B-5000 Namur, Belgium
{ivan.jureta, stephane.faulkner}@fundp.ac.be
[2] Information System Research Unit, University catholic of Louvain,
1 Place des doyens, B-1348 Louvain-la-Neuve, Belgium
kolp@isys.ucl.ac.be

Abstract. This paper proposes an agent-oriented metamodel that provides rigorous concepts for conducting enterprise modelling. The aim is to allow analysts to produce an enterprise model that precisely captures the knowledge of an organization and of its business processes so that an agent-oriented requirements specification of the system-to-be and its operational corporate environment can be derived from it. To this end, the model identifies constructs that permit capturing the intrinsic characteristics of an agent system such as autonomy, intentionality, sociality, identity and boundary, or rational self-interest; an agent being an organizational actor and/or a software component. Such an approach of the concept of agent allows the analyst to have a holistic perspective integrating human and organizational aspects to gain better understanding of business system inner and outer modelling issues. The metamodel has roots in both management theory and requirements engineering. It helps to bridge the gap between enterprise and requirements models proposing an integrated framework, comprehensive and expressive to both managers and software (requirements) engineers.

1 Introduction

Business analysts and IT managers have advocated these last fifteen years the use of enterprise models to specify the organizational and operational environment (outer aspects of the system) in which a corporate software will be deployed (inner aspects of the system) [1]. Such a model is a representation of the knowledge an organization has about itself or of what it would like this knowledge to be. This covers knowledge about functional aspects of operations that describe what and how business processes are to be carried out and in what order; informational aspects that describe what objects are to be processed; resource aspects that describe what or who performs these processes according to what policy; organizational aspects that describe the organizational architecture within which processes are to be carried out; and, finally, strategic aspects that describe why processes must be carried out. The specification of these key aspects of the core business of an enterprise is an effective tool to consider for gathering and eliciting software requirements. It may be used to [2, 3]:

M. Kolp et al. (Eds.): AOIS 2005, LNAI 3529, pp. 184 – 199, 2006.

- analyse the current organizational structure and business processes in order to reveal problems and opportunities;
- evaluate and compare alternative processes and structures;
- achieve a common understanding and agreement between stakeholders (e.g. managers, owners, workers) about different aspects of the organization;
- reuse knowledge available in the organization.

This paper proposes an integrated agent-oriented metamodel for enterprise modelling. The agent paradigm is a recent approach in software engineering that allows developers to handle the lifecycle of complex distributed and open systems required to offer open and dynamic capabilities in the latest generation enterprise software (see e.g. [4]).

The proposed metamodel takes inspiration from research works in requirements engineering frameworks (see e.g. [5-6]), management theory concepts found to be relevant for enterprise modelling (see e.g. [7-9]) and agent-oriented software engineering (see e.g. [4]). It leads to the reduction of the semantic gap between enterprise and requirements representations, providing a modelling tool that integrates the outer specification of the system together with its inner specification. Our proposal implicitly suggests a holistic approach to integrate human and organizational issues and gain better understanding of the representation of business processes and organizations representation. To this end, we introduce new concepts to enterprise modelling, related to authority, power and interest.

The rest of this paper is organized as follows. Section 2 describes the main concepts of our metamodel. Sections 3 and 4 detail some elements of the metamodel using the Z specification langage and discuss their relevance for enterprise modelling. Section 5 gives an overview of related works and Section 6 summarizes the results and points to further work.

2 An Agent-Oriented Enterprise Metamodel

The motivation of our proposal is to understand precisely the semantics of the organizational environment of the system and to produce an agent-oriented requirements specification for the software to build. The framework described in this section provides modelling constructs that permit the representation of the autonomy, intentionality, sociality, identity and boundary, as well as the rational self-interest of actors, i.e. agents in the real world and/or software agents. Actors are autonomous as their behaviour is not prescribed and varies according to their dependencies, personal goals and capabilities. They are intentional since they base their actions and plans on beliefs about the environment, as well as on goals they have to achieve. Being autonomous, actors can exhibit cooperative behaviour, resulting from similar goals and/or reciprocal dependencies concerning organizational roles they assume. The dependencies can either be direct or mediated by other organizational roles. Actors can have competing goals, which lead to conflicts that may result from competing use of resources. Actors have varying power and interest in the ways in which organizational goals contribute to their personal ones. Boundary and identity are closely related to power and interest of actors. We model variations in boundary and

identity as resulting from changes in power and interest since these vary with respect to the modifications in the roles an actor assumes and the dependencies involving these roles. Actors can act according to their self-interest, as they have personal goals to achieve. They have varying degrees of motivation to assume organizational roles, according to the degree of contribution to personal goals these roles have in achieving organizational ones. Actors apply plans according to the rationale described in terms of personal, organizational goals and capabilities. The rationale of our actors is not perfect, but bounded [10-11], since they can act based on beliefs that are incomplete and/or inconsistent with reality. We provide constructs such as AndOr relationships, non-functional requirements [4] etc. to evaluate alternative deployments of the software in the organizational environment.

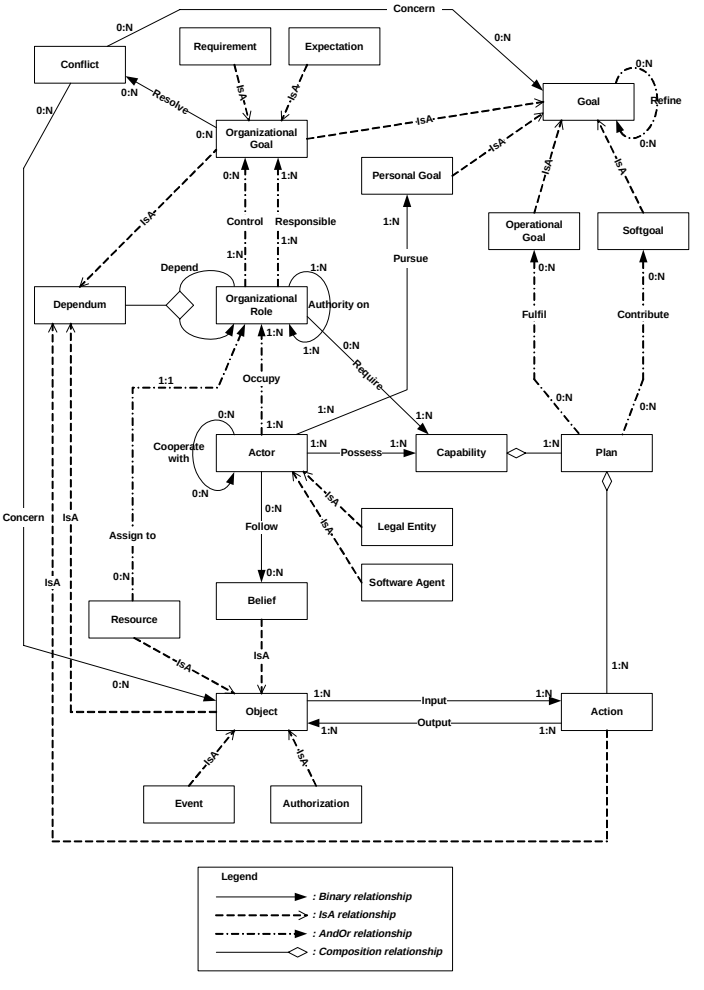

Fig. 1. The agent-oriented metamodel

Fig. 1 introduces the main entities and relationships of our metamodel. For clarity, we have subdivided it into five sub-models:

- *Organizational sub-model*, describing the actors of the organization, their organizational roles, responsibilities and capabilities.
- *Goals sub-model*, describing enterprise and business process purposes, i.e. what the actors are trying to achieve and why.
- *Conflict sub-model*, indicating inconsistencies in the business process.
- *Process sub-model*, describing how actors achieve or intend to achieve goals.
- *Objects sub-model*, describing non-intentional entities and assumptions about the environment of the organization and the business processes.

Due to a lack of space, the paper only details the organizational and goal sub-models, their integration and discusses their relevance for enterprise modelling. We first sketch the metamodel from the point of view of system developers and of organization managers.

2.1 Information System Development Perspective

The metamodel provides widely-used constructs for specifying the architecture of an agent-oriented information system: *Actors* are agents of the system. They *possess Capabilities* composed of *Plans*, each *Plan* representing a sequence of atomic *Actions*. When applying *Plans*, *Actors fulfil* or *contribute* to system *Goals*. *Actors follow Beliefs* which represent assertions about aspects of the organization and/or its environment. *Actions* can take *Objects* as *input* from the system or its environment. New *Objects* can be produced or existing ones modified by carrying out *Actions*, i.e. they can be *output* from *Actions*. *Objects* represent any thing of interest for the system: *Resources*, *Beliefs*, *Authorizations* or *Events.*

2.2 Management Perspective

The metamodel provides common terms used to describe an organization. *Organizational Roles* are *responsible* of *Organizational Goals*, which may be either *Operational* (i.e. can be actually fulfilled) or *Softgoals* (such as e.g. broadly specified business objectives). *Organizational Roles* can *depend* on one another for the provision of *Dependums* - *Actions*, *Objects*, or *Organizational Goals*. An *Actor*, being a *Human Actor* or a *Software Agent*, can *occupy Organizational Roles*, as long as it *possesses* the *required Capabilities* to do so. *Actors* exhibit intentional behaviour since they act according to *Goals* and *Beliefs* about their environment. Since *Beliefs* may be incoherent, and as they pursue *Personal Goals*, *Actors* can exhibit competitive behaviour. They will exhibit cooperative behaviour when they are responsible of identical *Organizational Goals*. *Actors* execute *Plans*, composed of *Actions*, in order to fulfil and contribute to *Goals*. By doing so, they comply with the responsibilities of *Organizational Roles* they *occupy*. As a matter of organizational policy, *Resources* in the organization are assigned to *Organizational Roles*. The allocation of Resources is determined by both *authority* among *Organizational Roles* and *Authorizations* that may be *input* or *output* of specific *Actions*.

Common ground between both points of view resides in the sense that the information system can be developed to automate some (part of) business processes (e.g. administrative tasks) or to radically modify ways in which *Goals* are fulfilled (e.g. reorganizing customer relationship management by deploying e-commerce facilities). The model provides an unambiguous representation serving both software staff and organization strategic management.

Primitives of our framework are of different types: meta-concepts (*Goal*, *Actor*, *Object* etc.), meta-relationships (*possess*, *require*, *pursue* etc.), meta-attributes (*Power*, *Interest*, *Motivation* etc.) and meta-constraints (e.g. "*an actor occupies a position if that actor possesses all the capabilities required to occupy it*").

All meta-concepts, meta-relationships and meta-constraints have the following mandatory meta-attributes:

- *Name*, which allows unambiguous reference to the instance of the meta-concept (e.g. "European Commission" for the Actor meta-concept).
- *Description*, which is a precise and unambiguous description of the corresponding instance of the meta-concept. The description should contain sufficient information so that a formal specification can be derived for use in requirements specifications for a future information system.

Fig.1 shows only meta-concepts and meta-relationships. Meta-attributes and meta-constraints are specified in the next sections using the Z state-based specification language [12, 13]. We use Z since it provides sufficient modularity, abstraction and expressiveness to describe in a consistent, unified and structured way an agent-oriented IS and the wider context in which it is used. It has a pragmatic approach to specifications by allowing a clear transition between specification and implementation of software [13]. In addition, it is widely accepted in the software development industry and has been used in large-scale projects.

3 Organizational Sub-model

The Organizational sub-model is used to identify the relevant *Actors* of the organization, the *Organizational Roles* they *occupy*, the *Capabilities* they *possess* and the *Dependum* for which *Actors depend on* one another.

3.1 Actor

Fig. 2 shows the Z formal specification of the *Actor* concept. The first part of the specification represents the definition of types. A given type defines a finite set of items. The *Actor* specification first defines the type Name (which represents the *Name* attribute) by writing [Name]. Such a declaration introduces the set of all names, without making assumptions about the type (i.e. whether the name is a string of characters and numbers, or only characters, etc.). Note that the type Actor_Type is defined as being either a Human_Actor or a Software_Agent. Defining types in such way indicates either that further detail about the type would not add significant descriptive power to the specification or that a more elaborate internal representation is not required.

More complex and structured types are defined with schemata. A schema groups a collection of related declarations and predicates into a separate namespace or scope. The schema in Fig. 2 is entitled ***Actor*** and is partitioned by a horizontal line into two sections: the declaration section above and the predicate section below the line. The declaration section introduces a set of named, typed variable declarations. The predicate section provides predicates that constrain values of the variables, i.e. predicates are used to represent constraints. In order to clarify the Z formal specifications of the concepts, we will refer in the text to specific Z schema predicates by using identifiers placed left of the schema in the form e.g. "(c1)" to refer to predicate, i.e. constraint (c1) of the schema.

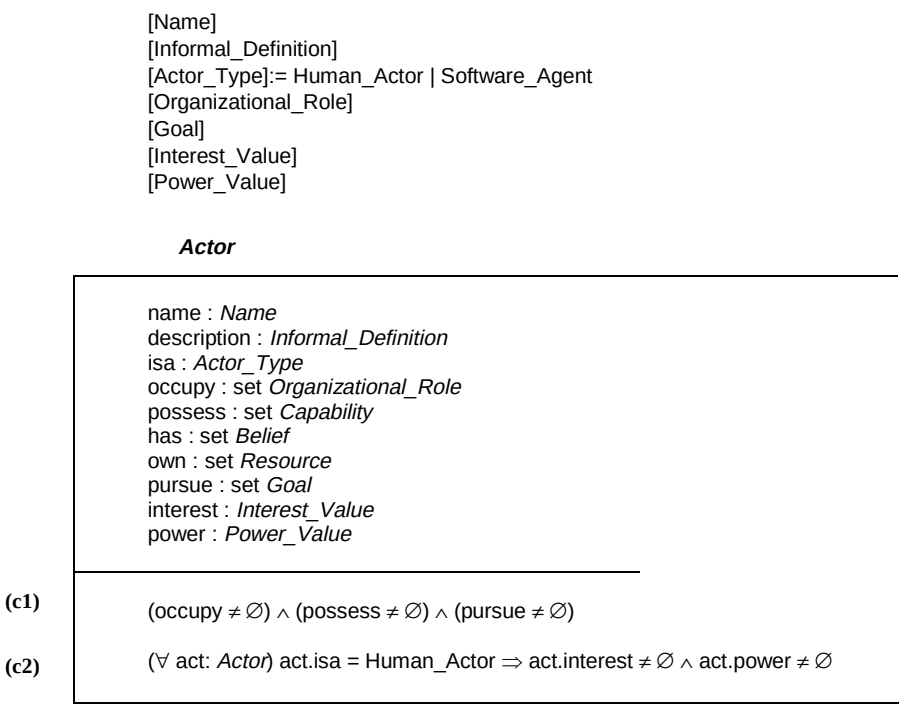

Fig. 2. Formal specification of the *Actor* concept

An Actor applies *Plans* (which are part of his *Capabilities*) to *fulfil* and/or *contribute* to *Organizational Goals* for which the *Organizational Role* he/she *occupies* is *responsible* and *Personal Goals* he/she *pursues* (i.e. wishes to achieve). As the *Actor* exists in a changing environment, it *follows Beliefs* about the environment in order to adapt its behaviour to environmental circumstances.

An *Actor* is either a *Human Actor* or a *Software Agent*. A *Human Actor* is used to represent any person, group of people, organizational units or other organizations that are significant to the organization we are modelling, i.e. that have an influence on its resources, its goals etc. A *Software Agent* is used to represent a software component

of an information system(-to-be). An *Actor* can *cooperate with* another *Actor* to fulfil and/or contribute to *Organizational Goals* common to the *Organizational Roles* that each of these *Actors occupies*.

Besides standard meta-attributes, a *Human Actor* is characterized with two specific meta-attributes: *Interest* and *Power*. *Interest* is the degree of satisfaction of an actor to see *Organizational Goals* positively contributing to its *Personal Goals*. *Power* is the degree to which the actor is able to modify the objectives of the organization or its business processes through its *Capabilities*. For instance, when automating a business process, the values of *Interest* and *Power* meta-attributes of *Human Actors* change: in the new configuration of the process, some actors will gain decision power while maintaining the same level of interest; others that previously benefitted from high power in the initial process structure might become less powerful. It is crucial to take these changes into account when eliciting software requirements. It may lead otherwise to introducing *Goals* not identified during the initial requirements analysis, and/or changing *Priority* of already specified *Goals*. *Interest* and *Power* help to find *Human Actors* that will play a crucial role in the software-to-be. For example, focus in some business process might shift to *Human Actors* that were not considered very significant during the inception phase and whose needs were not specified in depth. This would result in that these now crucial processes would not be fully exploited and would lead to the overall failure of the requirements specification efforts.

3.2 Organizational Role

An *Organizational Role* is an abstract characterization of expected behaviour of an *Actor* within some specified context of the organization. An *Actor* can *occupy* multiple roles and a role can be *occupied* by multiple *Actors*.

From an agent orientation perspective, *Organizational Roles* provide the building blocks for agent social systems and the requirements by which agents interact. The concept of *Organizational Role* is important to abstractly model the agents in multi-agent systems and helpful to manage its complexity without considering the concrete details of agents (e.g. implementation architectures and technologies).

Fig. 3 shows the Z formal specification of the *Organizational Role* concept. Each *Organizational Role requires* a set of *Capabilities* to fulfil or contribute to *Organizational Goals* for which it is *responsible*. An *Actor* can *occupy* the *Organizational Role* only if it *possesses* the required *Capabilities* (c4)[1]. In addition to entering *Organizational Roles*, *Actors* should be able to leave roles at runtime. The attribute *Leave Condition* is used to specify the *Belief* that has to be true in order for the *Actor* to leave the *Organizational Role* (c5).

Organizational Roles are *responsible* for *Organizational Goals* (c6) and can *control* their *fulfilment*. In case an *Organizational Goal* has been fulfilled, the *Actor*, occupying the *Organizational Role* that *controls* that *Goal*, executes a *Plan* in which an *Action output*s a new *Belief* to mark the goal fulfilment (c7). This control procedure requires that a single *Actor* can never occupy distinct *Organizational Roles* that are *responsible* of and *control* the fulfilment of the *Organizational Goal* (c8).

[1] To clarify the formal specifications, we embed the comments on predicates between two "//" signs.

[Goal_Control_Status]:= Fulfilled | Unfulfilled
[Belief]

Organizational Role

name : *Name*
description : *Informal_Definition*
require : set *Capability*
leave_condition: set *Belief*
responsible : set *Goal*
control : set (*Organizational_Goal*, *Goal_Control_Status*)
authority_on : set *Organizational_Role*

(c3) (require $\neq \emptyset$) $\wedge$ (leave_condition $\neq \emptyset$) $\wedge$ (responsible $\neq \emptyset$)

(c4) ($\forall$ act: *Actor*; r: *Organizational_Role*)
r $\in$ act.occupy $\Rightarrow$ r.require $\subset$ act.possess
//An Actor *act* that occupies the Organizational Role *r* possesses the Capabilities required by the Organizational Role *r*.//

(c5) ($\forall$ act: *Actor* ; r: *Organizational_Role*)
act.has $\subset$ r.leave_condition $\Rightarrow$ r $\notin$ act.occupy
//If the Leave Condition is true, than the Actor *act* no longer occupies the Organizational Role *r*.//

(c6) ($\forall$ r: *Organizational_Role* ; g: *Goal*)
g $\in$ r.responsible $\Rightarrow$ g.sec_isa = Organizational_Goal
//If Organizational Role *r* is responsible of Goal *g*, then *g* is an Organizational Goal.//

(c7) ($\forall$ r: *Organizational_Role*; g: *Goal*)
(g.prim_isa = Operational_Goal $\wedge$ g.sec_isa = Organizational_Goal
$\wedge$ g $\in$ r.control $\wedge$ g.status = Fulfilled)
$\Rightarrow$ ($\exists$ b!: *Belief*) (g.status = Fulfilled) $\in$ b.term $\wedge$ (g, Fulfilled) $\in$ r.control)
//If an Organizational Operational Goal *g* is fulfilled, then the Organizational Role *r* which controls the fulfilment of *g* outputs a new Belief *b* which indicates that the Goal *g* has been fulfilled.//

(c8) ($\forall$ r_1, r_2: *Organizational_Role* ; g: *Goal* ; a_1, a_2: *Actor*)
(g.sec_isa = Organizational_Goal $\wedge$ g $\in$ r_1.responsible $\wedge$ g $\in$ r_2.control $\wedge$ $r_1 \neq r_2$ $\wedge$ $r_1 \in$ act.occupy $\wedge$ $r_2 \in$ act.occupy) $\Rightarrow$ $a_1 \neq a_2$
//There can be no Actor *a* which occupies both the Organizational Role r_1 which is responsible for Organizational Goal *g*, and the Organizational Role r_2 which controls the fulfilment of Organizational Goal *g*.//

Fig. 3. Formal specification of the *Organizational Role* concept

Organizational Roles can have different levels of authority. Consequently, an *Organizational Role* can have *authority on* other *Organizational Roles*. The *authority on* relationship specifies the hierarchical structure of the organization. For instance, in the context of multi-agent systems, it can be used to define security policies that differ according to authority attributed to software agents.

3.3 Capability

A Capability specifies the behaviours that Organizational Roles should have in order to be responsible for or to control their Organizational Goals. An Actor possesses Capabilities. The formal specification in Fig. 4 shows that a Capability can be structured as a set of Plans and/or other Capabilities. This increases system modularity since libraries of capabilities can be built up and then combined to provide complex functionalities.

When exploring possible alternative business processes or organizational structures, newly identified Organizational Roles can require Capabilities that no Actor possesses. These Capabilities have to be confronted to those available in the organization (Capabilities that the Actors possess, see (c10)), in order to evaluate the proposed alternatives with respect to the current Roles and the way they use existing Capabilities. This is significant to determine which and how the proposed Capabilities and Roles will be finally introduced through the system-to-be. The availability of a Capability is formally expressed through the availability attribute, as indicated in the Capability schema.

[Cap_Atom]:= Plan | Capability
[Cap_Availability]:= Available | Unavailable

Capability

name : *Name*
description : *Informal_Definition*
composed_of : set *Cap_Atom*
availability : *Cap_Availability*

(c9) composed_of $\neq \emptyset$

(c10) ($\forall$ cap: *Capability*)
$\exists$ act: *Actor* ; cap $\in$ act.possess $\Rightarrow$ cap.availability = available
//If there is some Actor *act* that possesses Capability *cap*, then *cap* is available.//

Fig. 4. Formal specification of the *Capability* concept

3.4 Dependum

An *Organizational Role depends* on another *Organizational Role* for a *Dependum*, so that the latter may provide the *Dependum* to the former. A *Dependum* can be an *Organizational Goal*, an *Object* or an *Action*. In the *depend* meta-relationship, the *Organizational Role* that depends on is called the depender and the *Organizational Role* being depended upon is called the dependee.

We define the following dependency types:

- *Organizatonal Goal*-dependency: the depender *depends* on the dependee to *fulfil* and/or *contribute* to an *Organizational Goal*. The dependee is given the possibility to choose *Plans* through which it will *fulfil* and/or *contribute* to the *Organizational Goal*.

- *Action*-dependency: The depender *depends* on the dependee to accomplish some specific *Action*.
- *Object*-dependency: The depender *depends* on the dependee for the availability of an *Object*.

[Dependum_Type]:= Organizational_Goal | Object | Action

Dependum

name : *Name*
description : *Informal_Definition*
type : *Dependum_Type*
depender : set *Organizational_Role*
dependee : set *Organizational_Role*

(c11) $(type \neq \emptyset) \wedge (depender \neq \emptyset) \wedge (dependee \neq \emptyset)$

(c12) $(\forall$ d: *Dependency* ; dpd: *Dependum* ; r_1, r_2: *Organizational_Role*) $r_1 \neq r_2 \wedge (d \equiv r_1 \times dpd \times r_2) \Rightarrow (depender = r_2 \wedge dependee = r_1)$

(c13) $(\forall$ d: *Dependency* ; dpd: *Dependum* ; r_1, r_2: *Organizational_Role*) $r_1 \neq r_2 \wedge (d \equiv r_1 \times dpd \times r_2) \wedge (dpd.type = Authorization) \Rightarrow r_1 \in r_2.authority\_on$
//If the Dependum is an Authorization, then Dependee r_2 has authority on Depender r_1.//

(c14) $(\forall$ obj: *Object* ; a_1, a_2: *Actor* ; cap_1, cap_2: Capability ; pl_1, pl_2: *Plan* ; $actn_1$, $actn_2$: *Action* ; r_1,r_2: *Organizational_Role*)
$(a_1 \neq a_2 \wedge cap_1 \neq cap_2 \wedge pl_1 \neq pl_2 \wedge actn_1 \neq actn_2 \wedge (actn_1 \in pl_1.composed\_of \wedge pl_1 \in cap_1.composed\_of \wedge cap_1 \in a_1.possess) \wedge (actn_2 \in pl_2.composed\_of \wedge pl_2 \in cap_2.composed\_of \wedge cap_2 \in a_2.possess) \wedge obj \in actn_1.postcondition \wedge obj \in actn_2.input \wedge r_1 \in a_1.occupy \wedge r_2 \in a_2.occupy \wedge \{r_1,r_2\} \notin \{a_1.occupy \cap a_2.occupy\}) \Leftrightarrow (\exists$ dm: *Dependum* $\wedge$ dm.type = Object $\wedge$ dm.name = obj.name $\wedge$ dm.depender = $r_2 \wedge$ dm.dependee = r_1)
//Suppose that there are two different Actors a_1 and a_2 that respectively occupy two different Organizational Roles r_1 and r_2. These Actors possess respectively two different Capabilities cap_1 and cap_2, which respectively contain distinct Plans pl_1 and pl_2. These plans enable them to execute respectively the distinct Actions $actn_1$ and $actn_2$. If Action $actn_1$ has Object *obj* in its postcondition, and Action $actn_2$ outputs *obj*, then Organizational Role r_2 depends on the Organizational Role r_1 to provide the Object *obj*.//

Fig. 5. Formal specification of the *Dependum* concept

Object dependency allows us to represent any specialization of the *Object* concept as a *Dependum*. For example, an *Organizational Role* r_1 might *depend* on another *Organizational Role* r_2 for an *Authorization*. This has implications on the *authority on* relationship, as this dependency means that r_2 must have *authority on* r_1 (c13).

The constraint (c14) in Fig. 5 shows that the existence of an *Object Dependum* among *Organizational Roles* has implications on the *Input* and *Postcondition* of *Actions* accomplished by *Actors* that *occupy* these *Organizational Roles*. This

constraint provides a mapping rule between *depend* and *input*/*output* relationships. Its interest (c14) is twofold:

- If we know *Object* dependencies existing among several organizational roles, we can derive the activity diagram and the collaboration diagram (such as the ones in UML) without difficulties: actions that are related by dependencies (through their respective inputs/outputs) can be either sequential or parallel, which is sufficient to define the activity diagram. In addition, we know the actors that need to execute actions, as we know the organizational roles involved in dependencies.
- If we know the sequence of activities in a process, we can derive the dependencies among roles that participate in the realization of the process. Dependencies can then be analysed for vulnerabilities and alternative process structures can be evaluated.

This is an important difference of our approach compared to *i** [5]: we can use the link established between dependencies and actions in e.g. analyzing simultaneously the dependencies among organizational roles and the behavioural aspects of the process being analysed in terms of sequence of actions that compose it. This constraint makes it possible to combine the strengths of the *i** dependency representation, notably in terms of strategic dependency analysis among the process' organizational roles, with the analysis of the realization of the process as a series of sequential and/or parallel actions, that can be realized using e.g. UML activity and collaboration diagrams or scenario-based approaches.

4 Goals Sub-model

A *Goal* describes a desired or undesired state of the environment. The environment is the context in which actors live and interact with other actors. A state of the environment is described through the states of *Objects* (*Beliefs*, *Resources* etc.).

In addition to standard attributes, a *Goal* is characterized by the optional *Priority* attribute [14], which specifies the extent to which the goal is optional or mandatory. The values and the measurement of priority are domain specific.

To support qualitative and formal reasoning about goals, we classify them along two axes: *Operational Goals vs. Softgoals* and *Organizational Goals vs. Personal Goals*. In addition, we use patterns to specify the temporal behaviour of *Goals*. These classifications are treated in more detail below.

Operational Goal vs. Softgoal. An *Operational Goal* describes a desired or undesired state of the environment that can be achieved by applying *Plans*. An *Operational Goal* has been *fulfilled* if the state of the environment described by the *Operational Goal* has been achieved by a Plan. An *Operational Goal* has *State* and *Status* optional attributes (see Fig. 6). *State* describes the state of the environment in which the *Operational Goal* is fulfilled (c15). *Status* indicates whether the *State* of the *Operational Goal* has been achieved, i.e. whether the *Goal* has been fulfilled or not (c16).

A *Softgoal* also describes a desired or undesired state of environment, but its fulfilment criteria (i.e. how to achieve the desired state) may not be formally specified. A consequence of this is that *Plans* that are otherwise applied to *fulfil*

[Primary_Goal_Type]:= Operational_Goal | Softgoal
[Secondary_Goal_Type]:= Organizational_Goal | Personal_Goal
[Org_Goal_Type]:= Requirement | Expectation
[Goal_Pattern]:= Achieve | Cease | Maintain | Avoid
[Object]:= Resource | Authorization | Belief | Event
[Goal_Status]:= Fulfilled | Unfulfilled
[Refinment_Alternative]
[Priority_Value]
[Conflict]

Goal

name : *Name*
description : *Informal_Definition*
prim_isa : *Primary_Goal_Type*
sec_isa : *Secondary_Goal_Type*
org_isa : *Org_Goal_Type*
pattern : *Goal_Pattern*
state : set *Object*
status : *Goal_Status*
refined_by : set *Refinement_Alternative*
priority : *Priority_Value*
resolve : set *Conflict*

(c15) (∀ g: *Goal*) g.prim_isa = Operational_Goal ⇒ g.state ≠ ∅
//If Goal *g* is an Operational Goal, then *g* must have a specified state, i.e. the environment in which *g* is fulfilled must be specified as a set of Objects.//

(c16) (∀ g: *Goal*) g.prim_isa = Operational_Goal ∧ ∃ oset = {ob_1,…,ob_n : *Object*} ∧ g.state ⊆ oset ⇒ g.status = Fulfilled
//If there is a set of Objects *oset*, such that the state of Goal *g* is a subset of *oset*, then *g* is fulfilled.//

(c17) (∀ g: *Goal*) g.sec_isa = Organizational_Goal ⇔ g.org_isa ≠ ∅
//If the Goal *g* that is an Organizational Goal, then *g* must be either a Requirement or an Expectation.//

(c18) (∀ g: *Goal*; r: *Organizational_Role* ; act: *Actor*)
(g.sec_isa = Organizational_Goal ∧ r ∈ act.occupy ∧ g ∈ r.responsible ∧ act.isa = Software_Agent) ⇒ g.org_isa = Requirement
//An Organizational Goal *g* is a Requirement if there is some Software Agent Actor *act* which occupies the Organizational Role *r* which in turn is responsible for *g*.//

(c19) (∀ g: *Goal*; r: *Organizational_Role* ; act: *Actor*)
(g.sec_isa = Organizational_Goal ∧ r ∈ act.occupy ∧ g ∈ r.responsible ∧ act.isa = Human_Actor) ⇒ g.org_isa = Expectation
//An Organizational Goal *g* is an Expectation, if there is a Human Actor *act* which occupies an Organizational Role *r* which in turn is responsible for *g*.//

(c20) (∀ g: *Goal*) g.sec_isa ≠ Organizational_Goal ⇒ g.resolve = ∅
//If Goal *g* is not an Organizational Goal, then *g* cannot resolve Conflicts.//

Fig. 6. Formal specification of the *Goal* concept

Operational Goals can only *contribute* (positively or negatively) to *Softgoals*. For example, "increase customer satisfaction" and "improve productivity of the workforce" are *Softgoals*.

Organizational Goal vs. Personal Goal. An *Organizational Goal* describes the state of the environment that should be achieved by cooperative and coordinated behaviour of *Actors*. An *Organizational Goal* is either a *Requirement* or an *Expectation* (c17). A *Requirement* is an *Organizational Goal* under the responsibility of an *Organizational Role occupied* by a *Software Agent* (c18). An *Expectation* is an *Organizational Goal* under the responsibility of an *Organizational Role occupied* by a *Human Actor* (c18). This distinction between a requirement of the information system and the expectation of its human users contributes to the successful accomplishment of a process that generally involves interaction among them. *Organizational Goals* can solve *Conflicts* (c20) by specifying the state of the environment in which the *Conflicts* cannot be true.

A *Personal Goal* describes the state of the environment that an *Actor pursues* individually (i.e. without cooperative and coordinated behaviour). It can require competitive behaviour with other *Actors*.

We distinguish what is expected from the participation of the *Actor* in the process (through the *Organizational Role* it *occupies*) from what the *Actor* expects from its participation in the process (*fulfilment* of or *contribution* to its *Personal Goals*). In reality, consistency between the *Organizational Goals* and *Personal Goals* is not necessarily ensured. Consequently, it is important to reason about *Conflicts* that may arise between *Personal* and *Organizational Goals*, as well as about the degree to which an *Organizational Goal* assists in the pursuit of *Personal Goals*.

Temporal Behaviour of Goals. A behavioural pattern is associated with each *Goal*. The possible patterns are: *achieve*, *cease*, *maintain* and *avoid* [6]. For example, organizations tend to *avoid* "conflict of interest" (*Softgoal*) and *achieve* "replenish stock" (*Operational Goal*). When we associate a pattern to a *Goal*, we restrict the possible behaviour of the *Actors* concerning the *Goal*: *achieve* and *cease* generate behaviour, whereas *maintain* and *avoid* restrict behaviour.

5 Related Works

Process-oriented approaches such as Activity Diagrams, DFDs, IDEF0, workflows (see e.g. [11, 15-17]) describe an enterprise's business processes as sets of activities. Strong emphasis is put on the activities that take place, the order of activity invocation, invocation conditions, activity synchronization and information flows. Among these approaches, workflows have received considerable attention in the literature. In such a process-oriented approaches, agents have been treated as a computational paradigm, with a focus on the design and implementation of agent systems, not on the analysis of enterprise models.

Actor-oriented approaches emphasize the analysis and specification of the role of the actors that participate in the process [18]. The *i** modelling framework [5] has been proposed for business process modelling and reengineering. Processes, in which information systems are used, are viewed as social systems populated by intentional

actors that cooperate to achieve goals. The framework provides two types of dependency models: a strategic dependency model used for describing processes as networks of strategic dependencies among actors and the strategic rationale model used to describe each actor's reasoning in the process, as well as to explore alternative process structures. The diagrammatic notation of *i** is semi-formal and has proved useful in requirements elicitation (see e.g. [8, 19-20]). In this context, actor-oriented approaches provide significant advantages over other approaches: agents are autonomous, intentional, social etc. [4], which is of particular importance for the development of open distributed information systems in which change is ongoing. However, actors have served mostly as requirements engineering modelling constructs for real-world agents, without assuming the use of agent software as the implementation technology nor the use of organizational actors for enterprise modelling.

Goal-oriented approaches focus on goals that the information system or a business process should achieve. Frameworks like KAOS [6, 21] provides a formal specification language for requirements engineering, an elaboration method and meta-level knowledge used for guidance while the method is applied [22]. The KAOS specification language provides constructs for capturing the various types of concepts that appear during requirements elaboration. The elaboration method describes steps (i.e. goal elaboration, object capture, operation capture etc. [22]) that may be followed to systematically elaborate KAOS specifications. Finally, the meta-level knowledge provides domain-independent concepts that can be used for guidance and validation in the elaboration process.

Enterprise Knowledge Development (EKD) [18] is used primarily in modelling of business processes of an enterprise. Through goal-orientation, it advocates a closer alignment between intentional and operational aspects of the organization and links re-engineering efforts to strategic business objectives. EKD describes a business enterprise as a network of related business processes, which collaboratively realize business goals. This is achieved through several sub-models: an enterprise goal sub-model (expressing the causal structure of the enterprise), an enterprise process sub-model (representing the organizational and behavioural aspects of the enterprise) and an information system component sub-model (showing information system components that support the enterprise processes) [18]. Agents appear in the EKD methodology but without explicit treatment of their autonomy and sociality [4]. In KAOS, agents interact with each other non-intentionally, which reduces the benefits of using agents as modelling constructs.

6 Conclusion

Modelling the organizational and operational context within which a software system will eventually operate has been recognized as an important element of the engineering process (e.g. [1]). Such models are usually founded on primitive concepts such as those of *actor* and *goal*. Unfortunately, no specific enterprise modelling framework really exists for engineering modern corporate IS. This paper proposes an integrated agent-oriented metamodel for enterprise modelling. Moreover, our approach differs primarily in the fact that it is founded on ideas from in requirements

engineering frameworks, management theory concepts found to be relevant for enterprise modelling and agent oriented software engineering.

We have only discussed here the concepts that we consider the most relevant at this stage of our research. Further classification of, for instance, goals is possible and can be introduced optionally into the metamodel. For example, goals could be classified into further goal categories such as Accuracy, Security and Performance. We also intend to define a strategy to guide enterprise modelling using our metamodel as well as to define a modelling tool à la Rational Rose to visually represent the concepts.

References

1. Castro J., Kolp M. and Mylopoulos J.: Towards Requirements-Driven Information Systems Engineering: The Tropos Project. In Information Systems (27), Elsevier, Amsterdam (2002)
2. Koubarakis M., Plexousakis D.: A formal framework for business process modelling and design. Information Systems 27 (2002). 299-319
3. Bernus P.: Enterprise models for enterprise architecture and ISO9000:2000. Annual Reviews in Control 27 (2003) 211–220
4. Yu E.: Agent-Oriented Modelling: Software Versus the World. Proceedings of the Agent-Oriented Software Engineering AOSE-2001 Workshop, Springer Verlag (2001)
5. Yu E.: Modelling Strategic Relationships for Process Reengineering. Ph.D. thesis, Dept. of Computer Science, University of Toronto (1994)
6. Dardenne A., van Lamsweerde A., Ficklas S.: Goal-directed requirements acquisition. Sci. Comput. Programming 20 (1993) 3-50
7. Brickley J.A., Smith C.W., Zimmerman J.L.: Managerial Economics and Organization Architecture. McGraw-Hill Irwin 2nd ed. (2001)
8. Faulkner S., Kolp M., Coyette A., Tung Do T.: Agent-Oriented Design of E-Commerce System Architecture. Proceedings of the 6th International Conference in Enterprise Information Systems Engineering, Porto (2004)
9. Simon H. A.: Rational Decision Making in Business Organizations. The American Economic Review, 69(4) (1979), 493-513
10. Johnson G., Scholes K.: Exploring Corporate Strategy, Text and Cases. Prentice Hall (2002)
11. Mentzas G., Halaris C., Kavadias S.: Modelling business processes with workflow systems: an evaluation of alternative approaches. International Journal of Information Management, 21 (2001) 123-135
12. Spivey J. M.: The Z Notation: A Reference Manual. 2nd Edition, Prentice Hall International (1992)
13. Bowen J.: Formal Specification and Documentation using Z: A Case Study Approach (1994)
14. Simon H.A.: Administrative Behavior : A Study of Decision-Making Processes in Administrative Organization. New York: The Free Press 3rd ed. (1976)
15. Kamath M., Dalal N.P., Chaugule A., Sivaraman E., Kolarik W.J.: A Review of Enterprise Process Modelling Techniques. In Prabhu V., Kumara S., Kamath M.: Scalable Enterprise Systems: An Introduction to Recent Advances. Kluwer Academic Publishers, Boston (2003)

16. Elmagarmid A., Du W.: Workflow Management: State of the Art Versus State of the Products. In Dogac A., Kalinichenko L., Tamer Ozsu M., Sheth A.: Workflow Management Systems and Interoperability. NATO ASI Series, Series F: Computer and Systems Sciences, 164, Springer Heidelberg (1998)
17. Sheth A.P., van der Aalst W., Arpinar I.B.: Processes Driving the Networked Economy. IEEE Concurrency, 7, (1999) 18-31
18. Kavakli V., Loucopoulos P.: Goal-Driven Business Process Analysis Application in Electricity Deregulation, Information Systems, 24 (1999) 187-207
19. Liu L., Yu E.: Designing information systems in social context: a goal and scenario modelling approach. Information Systems, 29 (2004) 187–203
20. Briand L., Melo W., Seaman C., Basili V.: Characterizing and Assessing a Large-Scale Software Maintenance Organization. In Procedings of the 17th International Conference on Software Engineering, Seattle, WA (1995)
21. van Lamsweerde A., Darimont R., Letier E.: Managing Conflicts in Goal-Oriented Requirements Engineering. IEEE Transactions on Software Engineering, Special Issue on Managing Inconsistency in Software Development (1998)
22. van Lamsweerde A.: The KAOS Metamodel –Ten Years After. Technical report, (2003).

Fragmented Workflows Supported by an Agent Based Architecture

C. Reese, J. Ortmann, S. Offermann, D. Moldt, K. Markwardt, and T. Carl

Department of Informatics, University of Hamburg
http://www.informatik.uni-hamburg.de/TGI

Abstract. Within the distributed systems area, specific software solutions are required due to the distribution of systems and their users in time and space. A key role can be seen in the coordination of processes in this context. Applications that support the work of people and enterprises within such settings need to support requirements such as flexibility, autonomy, coordination and synchronization. An example is the coordination of distributed interorganizational workflows. The dynamic adaptation of workflows is of particular importance in this area, since enterprises need to dynamically adapt to changes in market and to new demands. Another example for such a setting, where a workflow needs to be constantly adopted are virtual enterprises e.g. production workflows, where changing partnerships lead to changing requirements. Based on the formal modelling technique of high-level Petri nets we use workflow nets and an agent framework, both tool supported. This leads directly to an innovative architecture in this field combining several former approaches with respect to their advantages.

Keywords: Distributed workflows, agents, distributed workflow enactment service, high-level Petri nets, Capa, Renew.

1 Introduction

New business areas like interorganizational cooperations and virtual enterprises require new solutions due to the increasing interactions across organizational boundaries and the high dynamics in their interrelations. In these areas, workflow technology can help to coordinate the cooperation processes. Due to the different and additional requirements of interorganizational workflows, conventional workflow technology cannot be directly applied. To support interorganizational workflows, a workflow management system (WFMS) must support e.g. the privacy of internal subprocesses, security issues or the assignment and distribution of subworkflows.

Since the process perspective has been within the centre of interest, workflow management systems (WFMS) have had a revival in the context of the development of distributed applications. From a conceptual perspective, workflows can be enhanced through the agent concept. Agents offer a natural way to deal with open environments and are therefore of particular benefit for distributed systems as stated e.g. by Jennings [1]. Purvis [2], Blake [3] and others, as well

M. Kolp et al. (Eds.): AOIS 2005, LNAI 3529, pp. 200–215, 2006.

as former work of our group [4] wjp treated the combination of workflows and agents. Interorganizational workflows are treated by van der Aalst [5]. Here the concept of interorganizational workflows is discussed and illustrated in detail. Our approach focusses and stresses the possible autonomy of workflows, which is required when autonomous enterprises are involved in a common business process. This is of high relevance and can be achieved by using agents to encapsulate each autonomous part (fragment) of a workflow. Due to the autonomy of a fragment, other properties of agents, such as flexibility, adaptability, mobility, intelligence (social interaction, autonomous decision of internal decision problems with respect to the environment etc.) can be assigned to the workflow fragments. This is one of the particular strengths of our system, and distinguishes ours from other approaches for interorganizational workflow management systems (WFMS). This kind of systems allows for the development of distributed, concurrent, decentralized and flexible (business) applications.

The main contribution of this work is the presentation of an agent based workflow management system architecture for the distributed execution of workflows. The presented workflow management system architecture is of particular strength due to its agent orientation, its formal basis provided by Petri nets and its partial tool support. The distributed execution of workflows requires their splitting into deployable workflow fragments. The act of splitting complex workflows manually into suitable fragments could be an exhausting and error-prone task. For this reason, we introduce in this work a semi-automated approach to determine such workflow fragments.

These fragments, encapsulated by agents, are treated again as workflows and can be executed at different locations using different workflow enactment systems, which are the platforms of the agents. Therefore, we provide a concept for a distributed and concurrent workflow management system as well as a prototypical implementation based on the FIPA compliant agent framework Capa.

The paper is organized as follows. Each section covers both conceptual and technical issues. Section 2 introduces the underlying concepts, techniques and tools. The fragmentation algorithm for workflows is detailed in Section 3. The overall agent based architecture and distribution issues are explained in Section 4. The paper ends with a summary of the achieved results and a discussion about possibilities for further extension.

2 Conceptual and Technical Background

To obtain an overview, Fig. 1 shows the basic architecture of the system described here. It consists of a runtime environment established by Java and reference nets, a workflow (WF) engine and an agent environment. On top of this, we develop workflow agents based on the specifications of the WfMC (Workflow Management Coalition, see [6]). These agents provide the functionality of distributed agent based workflows to any application. In the following, the layers are described in more detail.

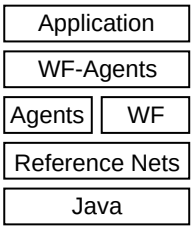

Fig. 1. Simple Architecture Overview

2.1 Reference Nets and Renew

For an introduction to reference nets, see [7,8]. Reference nets are an extension of the Coloured Petri net (CPN) formalism (for extensive introduction, see [9]) adding both the concept of nets-within-nets introduced by Valk [10] and the concept of synchronous channels (as first introduced in [11]). Additionally, reference nets allow to have multiple dynamically created net instances. Through an inscription language, reference nets allow for the execution of Java code from within a net when executed in the simulator.

Reference nets can be drawn, simulated and executed in the Renew tool (available at [12]), which is entirely implemented in Java. Offering true concurrency, different transitions of a Petri net can fire at the same time. While one task is executed, other parts of the application can continue. Along with the general expressive power of Petri nets, e.g. for concurrency, this makes reference nets a good choice for modelling and executing workflows.

2.2 Workflow Nets

The use of Petri nets in the workflow area has been thoroughly investigated (see [13]). Workflow patterns can be expressed by Petri nets (see [14]). Reference nets are especially suitable for defining a workflow due to their highly expressive power. Through the inscriptions on a transition, e.g. a legacy application can be called or a client can be asked to do something. Based on reference nets, an existing workflow plug-in for Renew by Jacob [15] is used as a basis to implement the concepts discussed here. This plug-in provides roles and several security features besides the general features of a workflow enactment service. It is based on a proposal for a concurrent, Petri net based workflow execution engine [16] and on the persistent Petri net execution engine presented in [17].

2.3 Agents

The technical agent framework Capa (Concurrent Agent Platform Architecture, see [18]) is based on the conceptual framework Mulan (Multi-agent nets). Capa is a special agent platform: The platform itself is implemented as an agent *containing* all agents residing on the platform, e.g. the FIPA compliant AMS and

DF agents (for FIPA, see [19]). This concept will be used to design WF agents in Section 4.2.

CAPA introduces the concept of *net agents* as an extensible architecture for agents. Such an abstract Petri net agent provides basic functionality such as sending and receiving messages. The behaviour is defined using protocol nets. *Protocol nets* are workflow-like nets. The interface to the containing agent enables explicit start and end points, incoming and outgoing information and access to the agent's knowledge base. The knowledge base provides adding, deleting, modifying and searching for entries in a key-value manner.

To describe agent interactions, we use AUML interaction protocol diagrams. The RENEW diagram plug-in provides tools for drawing of AUML interaction protocol diagrams. Skeletons of protocol nets for CAPA agents can be generated from those interaction diagrams. Interaction diagrams and the generation of protocol nets are discussed in detail by Cabac *et al.* [20].

2.4 The WfMC Components of a WFMS Within Our System

The structure model for workflow systems of the Workflow Management Coalition (WfMC, see [21]) defines six basic components of a WFMS, not repeated here (see [6]). We describe how we realize these in our system:

A *Process Definition Tool* is part of the existing RENEW workflow plug-in mentioned above. This is now extended to provide the possibility to define cut-off points for distribution within a workflow definition (this is detailed in Section 3).

A *Workflow Client Application* is also included into the existing plug-in and wrapped by an agent. *Invoked Applications* are wrapped by agents. Together with the concept of a *task agent* (defined in Section 4.2), this makes the distinction between client interactions, invoked applications and automatic tasks transparent to the workflow agent.

The *Workflow Enactment Service* is provided in our architecture by the existing workflow plug-in (see Section 2.2). This is wrapped by an agent as an agent platform (analogously to the CAPA architecture mentioned above) that communicates with the workflow engine agents components as *contained* agents. This makes the whole workflow enactment service encapsulated, gaining security, autonomy and mobility concepts.

The *workflow engines* are also wrapped by agents residing on the platform provided by the workflow enactment service. These coordinate the execution of workflow fragments on their platform.

Administration and monitoring is done by agents that gather information from components concerning running and finished tasks or problems. This provides a view to the system state as far as possible within distributed systems.

All agents are contained within a WFMS agent as a platform agent that provides the overall workflow management service.

2.5 Relation Between Workflows and Agents

For the simplest case, there is one workflow to be executed at one WFMS, locally. In this case, the agent that wraps that workflow is an interface to the

execution environment. In the next step, each workflow and workflow fragment is wrapped by an agent to transport it in an encapsulated way. As soon as it arrives at a location, it is executed using the same message interface. The conceptual advantages of agents, such as flexibility and autonomy, are only needed in an even more complex scenario, where a workflow or workflow fragment tries to reach its goals by interacting, negotiating, and by examining and judging the results. In case this does not satisfy its requirements, the workflow or workflow fragment can decide to autonomously change the circumstances, e.g. by trying another provider to produce a somehow "better" result. We provide the infrastructure for such a scenario by the design of workflows and workflow fragments as agents on their own.

Conceptually, through this design, two viewpoints on one application are combined by providing explicitly correspondent parts. Looking at an application as a workflow system, emphasizes some aspects like verifiability and structure control. Looking at the same application as a multi-agent system, emphasizes characteristics like autonomy, encapsulation and flexibility. So a certain application can be designed using both viewpoints at different stages of the process, but one technique, one architecture that represents both.

The technical realization of these concepts is easy because agents in Capa behave according to protocol nets that they instantiate according to their knowledge and goals. These are reference nets that provide the interface for a protocol net. Usually protocol nets have workflow-like structure. From the other side, workflows are represented through workflow nets, i.e. reference nets using special transitions and places for the special needs of a WFMS. Now a workflow or workflow fragment agent would instantiate a workflow net that also provides the interface as a protocol net. Access to the workflow definition would happen exclusively through the message interface of the agent and so gain security issues (at least conceptually, since we do not address security issues in the agent communication field). Mobility is provided by Capa.

2.6 Further Procedure

The overall way to our solution is now (see Fig. 2): firstly, we enable the definition of cut-off points within a workflow specification and with that we enable the distribution of workflow fragments; secondly, we map agent types to the WFMS components identified by WfMC and enrich these; finally, a workflow gets executed using the combined services of a workflow enactment service and a specific workflow agent.

3 Fragmentation of Workflow Nets

Within Petri nets, dependencies between net elements are locally defined by arcs. Only the neighbouring elements need to be examined and synchronized. The requirements for a workflow fragmentation are:

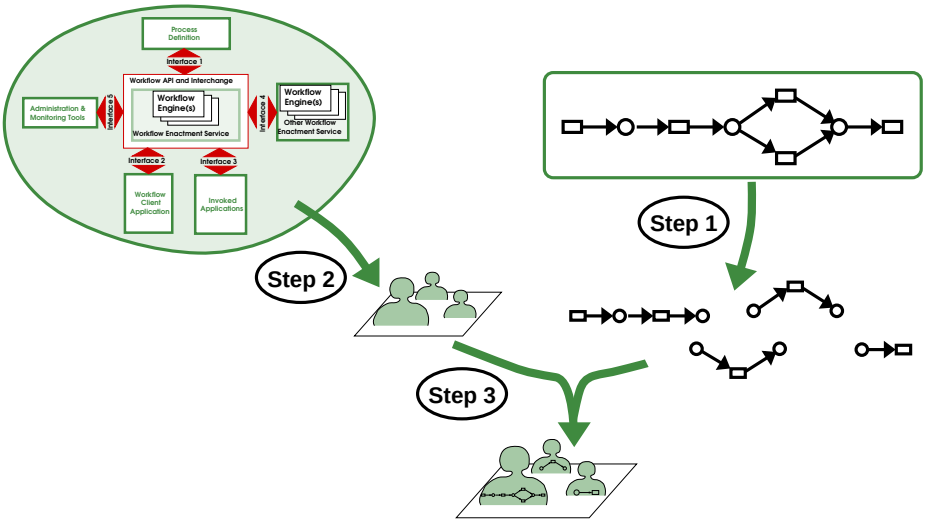

Fig. 2. Fragmentation and distribution of workflows

(1) Workflow fragments shall be independent except for synchronization at the borders.
(2) The fragmentation shall be arbitrary, in particular a XOR-split shall be possible and consequentially all major workflow patterns as described in [14].
(3) Each fragment shall have an arbitrary complex border, i.e. an arbitrary number of input and output arcs. Loops shall be possible.
(4) The semantics of the distributed workflow shall be the same as the semantics of the whole workflow, i.e. no additional elements shall be required to be drawn. This can be avoided by adding automatically some hidden refinements at fragment borders to implement synchronization functionality.

3.1 Border of Fragments

Basically three different types of border definitions are possible: split arcs, border places or border transitions. Split arcs do not limit the design and distribution of workflows, but the coordination costs are quite high, because synchronization and data transfer must be carried out with each search for bindings. The fragmentation at border places can be realized through a refinement of such a place where the synchronizing code is put. The border place can be seen as a distributed place with copies in each fragment. The change of the marking needs to be indivisible to prevent inconsistency. To define the border at transitions is the most intuitive definition of a border, because the transition is the active element within a net where data transfer happens anyway. In this case, there is no conflict possible and thus no distributed transaction is necessary. Once the firing is initiated, the action is completely isolated and may run concurrently

with other actions. The border transition would be a coarsening like the border place. Some workflow patterns, especially XOR, are not realizable with a transition split. Also, the same drawback as the split arcs holds here: the search for bindings would require costly remote communication. This is why we decide to use place bordered fragments.

3.2 Dividing a Workflow into Fragments

The designer needs to mark the intended border places in the workflow net. These operate as cutting points, where the transfers between fragments happen. Each workflow has one unambiguous starting point and one or more explicit endings. Using these fix-points, a fragmentation for the workflow can automatically be searched.

Each border place must satisfy the condition that at least a connection to two different fragments exists. If a border place connects only one fragment with itself, this should produce a warning because the intention of the programmer to generate fragments cannot be satisfied.

The following algorithm can be used for fragmentation and for a consistency check. It is implemented within the WF Agents plug-in. The algorithm is illustrated in Fig. 3: an example net and the resultant nets are shown. Transitions with bold borders are task transitions (`task1`, `2`, and `3`), places with arrows are distributed places (`dp_1`, `2`, and `3`).

Input: A workflow net with predefined border places.
(1) Transitions with `start` or `end` inscription may not have incoming or outgoing arcs, respectively. Otherwise the net is inconsistent.
(2) Transitions with `start` or `end` must be connected each to exactly one border place. Otherwise, the net is inconsistent.
(3) Regard all directed arcs as undirected and all transitions and places as unnamed nodes so border places become border nodes. Individually name all border nodes (no name conflicts in the example). Multiply border nodes according to the number of connected arcs and connect exactly one copy to each arc (In Fig. 3, this results in two instances of `dp_1` and three instances of `dp_2` and `dp_3`).
(4) For each unvisited border node search all connected border nodes in the undirected graph. Create a list with the names of connected nodes for each fragment (in the example, this results in five fragments, one of them containing node `start` and node `dp_1`).
(5) When no unvisited border nodes exist anymore, search for double border node entries in the list of each fragment. If a name occurs in one list and not at all in the other lists, this border node is inconsistent.
(6) Join the fragments containing the `start` and `end` nodes of the workflow and add a synchronized copy of each border node to obtain the control net.
(7) Regard nodes as places and transitions again. Put the fragments into individual nets and merge border places with common names across these nets: Mark the initial border place in the control net and give each concerned fragment a synchronized copy (a fragment is concerned if it contains a border place with the same name).

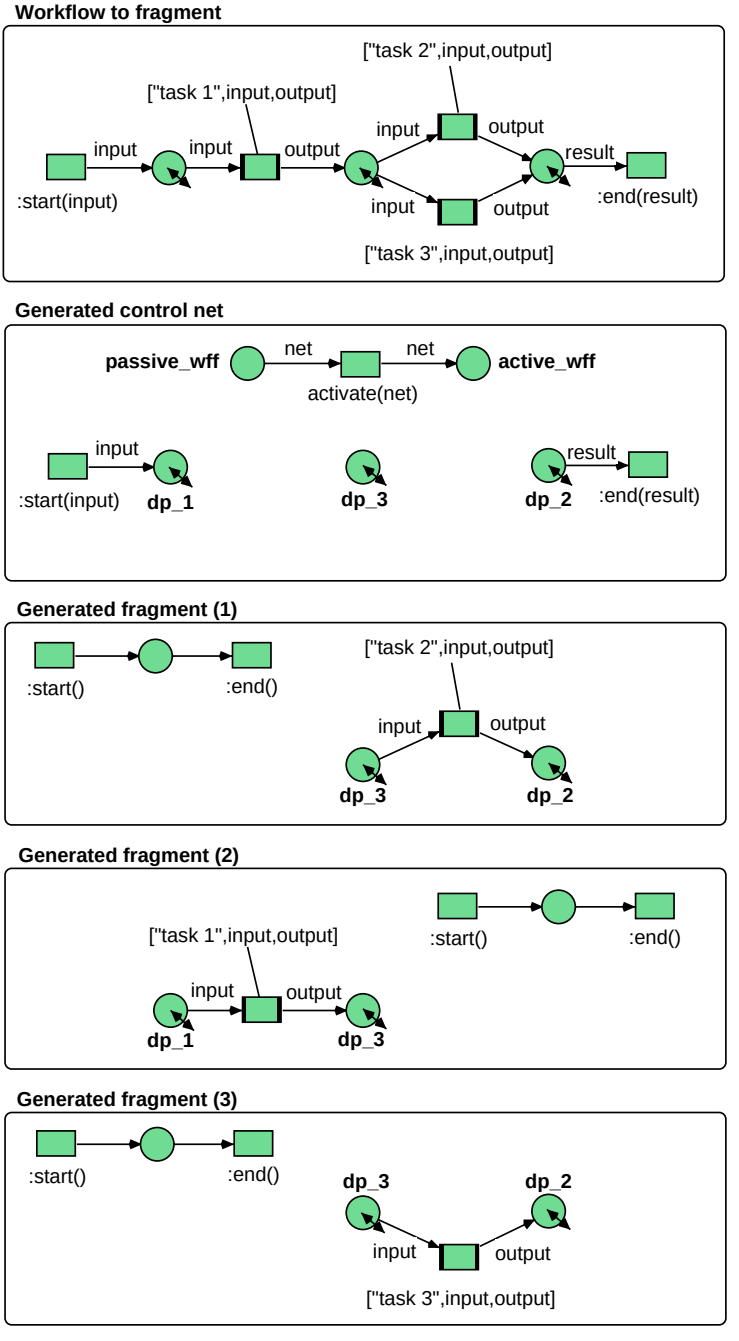

Fig. 3. Example workflow fragmentation

Result: An error message if the net is inconsistent, otherwise disjoint fragments (apart from border places), which, taken together, build the original net plus a control net. The control net holds `start` and `end` points of the workflow and all border places and their coordination. As soon as a token is put in one border place, all concerned workflow-fragments are activated, if they have a connected input arc. When the `end`-transition within the control net is activated, the workflow is considered finished.

3.3 Activation of Fragments and Termination of Workflows

Generally a workflow terminates once it has reached an explicitly defined end node.

For Petri nets, it basically holds that *a transition is activated* if all preconditions (markings, colors, guards...) are satisfied and *a Petri net is activated* if at least one transition is activated. Within Renew, a *net instance is passive* if there is no reference to it anymore, no transition is firing and no transition is activated. Otherwise, a garbage collector removes the net instance from the memory as soon as the space is needed. In the distributed case, it is conceptually not easy to keep track of references. Other than in the local case, an instantiated net can only be stopped explicitly by activating a special `end` transition. So a workflow fragment net is activated by the Workflow Fragment (WFF) agent and instantiated as a protocol net the first time one of its border places gets a token. It is deactivated only when the `end` transition in the control net is activated. The WF agent containing that control net informs all WFF agents, which, in turn, activate the `end` transitions of the WFF net instances.

Workflow nets should therefore be designed in such a way that nothing happens once the `end` transition was activated. In any case, this is part of the soundness-characteristic of a workflow net. Probably it is possible to prove this characteristic on generated workflow fragments.

4 Architecture for Agent Based Workflows

Our main motivation for a distributed workflow engine lies in the idea of cooperative work coordinated by distributed workflows [22]. Other approaches were motivated by load-balancing issues as in [23], such that workflows can be redistributed to other servers according to their load. The coordination of Web services is addressed in [3].

Furthermore, we focussed on the development of a FIPA-compliant framework, which is closely related to the standards proposed by the WfMC on the one hand and the use of reference nets as a formal executable basis for the modelling of the system on the other. Reference nets are used for the modelling of the system as well as for the modelling of the workflows. One major advantage of reference nets is their ability to directly execute Java code, which makes it possible to easily interact with a GUI or a program written in Java. Although other Petri net based architectures exist [2], to our knowledge, our architecture is the only one entirely based on reference nets with the benefits of easy Java integration,

a uniform architecture based on MULAN and a formalisms based on Petri nets enabling us to investigate issues such as fragmentation and distribution on an abstract level.

Our design of workflow agents building upon agent and workflow technology is depicted in Fig. 1, and is described in the following sections.

4.1 Plug-In Dependencies

The presented work makes use of several plug-ins for RENEW. The dependencies of the different plug-ins are shown in Fig. 4. RENEW provides a runtime environment and a GUI Plug-in. CAPA and the Workflow Plug-in depend on the RENEW simulator. CAPA provides optional GUI access (i.e. it can be used in a non-graphical environment).

The WF Agents Plug-in described in this paper depends on CAPA and on the WF Plug-in. Optional GUI access is provided. The direct dependency on the RENEW simulator results from the fragmentation of workflow nets, which requires extensions to basic net elements (i.e. the places including code for synchronization, as discussed in Section 3).

An Application using the system would depend on the WF Agents Plug-in and probably also on the CAPA Plug-in and the GUI Plug-in. These would form the runtime environment for that application.

4.2 Infrastructure

Each component of a WFMS can be mapped to an agent type (implementing this component) to form an agent based WFMS. We add a deployment agent and workflow agents that can hold a workflow as such.

Most of the defined agent types provide parts of the WF platform services. The task agents are domain specific service providers (cf. Fig. 5).

WFMS agent. The *Workflow Management Service* agent forms the platform of the WFMS containing all workflow specific agents. It provides the overall interface to the service of a WFMS by its contained agents.

WFES agent. The *Workflow Enactment Service* agent holds the main functionality of enacting workflows. It instantiates and contains workflow engine agents.

WFE agent. The execution of workflows is coordinated by *Workflow Engine* agents residing on the platform provided by the WFES agent. When a workflow is to be executed, this agent calls the service of a WF agent. All necessary communication for the execution is handled here.

WF-Cl agent. The *Workflow Client* agent replaces the client application from the underlying WFMS by shifting the communication level from Java to reference nets and from reference nets to agent messages. A participating user registers using this interface and receives task descriptions and work items according to his role. This is the workflow client application in the classical sense.

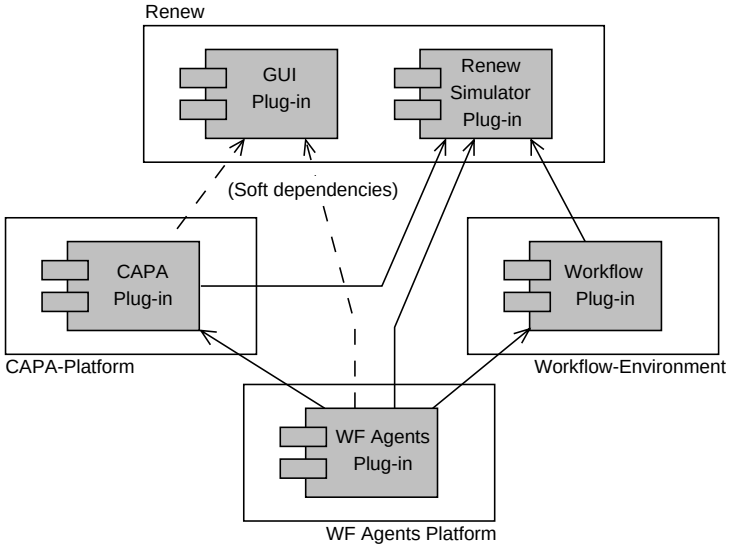

Fig. 4. Dependencies among Renew plug-ins

ADA agent. Within the *Administration Agent* the users of the system and their rights are managed.

Monitoring agent. This agent gathers information explicitly provided by the other agents concerning the execution state of the system. This agent can summarize gathered data and can act autonomously in exceptional situations.

Task agent. Task agents correspond to the "invoked application" in the WfMC model. They are arbitrary agents that can be provided by an application. Their supplied services are called by a task if this is required by a workflow. A Task agent is created by the WFES platform agent using application specific descriptions; it travels to its intended location, then executes its task by invoking some application.

WF and WFF agents. The workflows themselves reside as *Workflow* and *Workflow Fragment* agents on the platform provided by the WFES agent. The WF agent coordinates the WFF agents that are local or remote parts of the executed workflow.

Deployment agent. This agent realizes the configuration of the system. New workflows and roles can be configured here. It is not contained in the WFES agent platform.

Fig. 5 shows an exemplary infrastructure of the platform. The layers from which the platform is built are: the communication layer on HTTP basis at the bottom, which is provided by Capa. Above, the platform agents from Capa which provide basic services of a FIPA compliant platform are shown. These are the agents *Directory Facilitator* (DF), *Agent Management System* (AMS) and

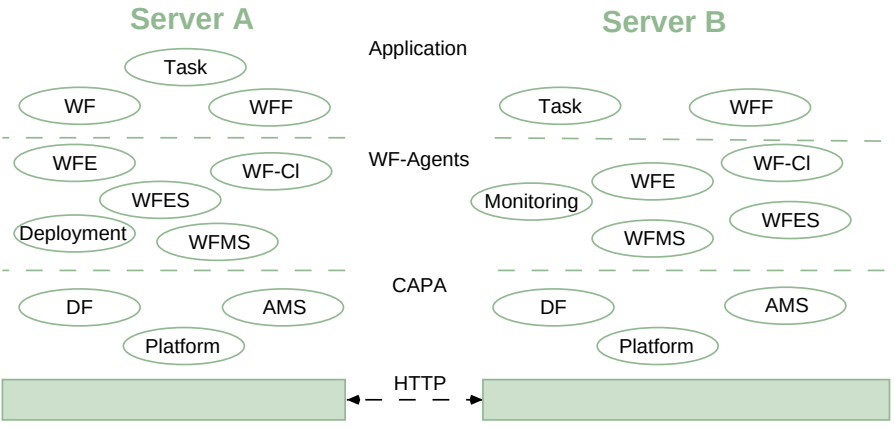

Fig. 5. Example infrastructure of WF platforms

the platform itself which is implemented as an agent in Capa. Again above are the agents of the WF agent platform. The *Workflow Management System* agent holds other agents analogous to the platform agent of Capa.

The agents are connected via the communication layer of Capa so that the platform components are loosely coupled. This gets us the advantage that components can independently be updated and started without affecting the whole system.

5 The Running System

In the following, some aspects of the running system are discussed.

Distribution and Execution. After the fragmentation, each fragment is encapsulated within one WFF (*Workflow Fragment*) agent. In terms of the agent framework Capa this means that the WFF agent runs a net instance of a workflow fragment net as a protocol net. The WFF agents are transferred to their execution platform (WFES agent), either through external channels or migrating or, third possibility, by starting the agents remotely. If a workflow is executed, the fragments must be located through a directory service but the WFF agents should provide their service to the associated WF agent only. To reach this, the involved agents must know each other. To make recognition possible, each fragment is signed and this signature is registered. Additionally, the WFF agents hold the signature of the original workflow to ensure authorization.

A workflow is started by calling the service of a WFES (workflow enactment service) agent which provides the services of contained WF agents. The WF agent searches for service providers for all fragments of the workflow. More than one provider for a certain fragment is possible in the case when more than one WF

agent was started for this workflow. The fragments are instantiated and activated according to the control net contained in the WF agent. As long as a fragment is not yet activated, it is possible to exchange the service provider. Within the control net the workflow is actually started and the first fragment is activated.

Synchronization between fragments is needed only at the border places. This is realized with a distributed lock i.e. a mechanism that ensures a consistent state for shared resources. Only the current owner of a lock may perform changes.

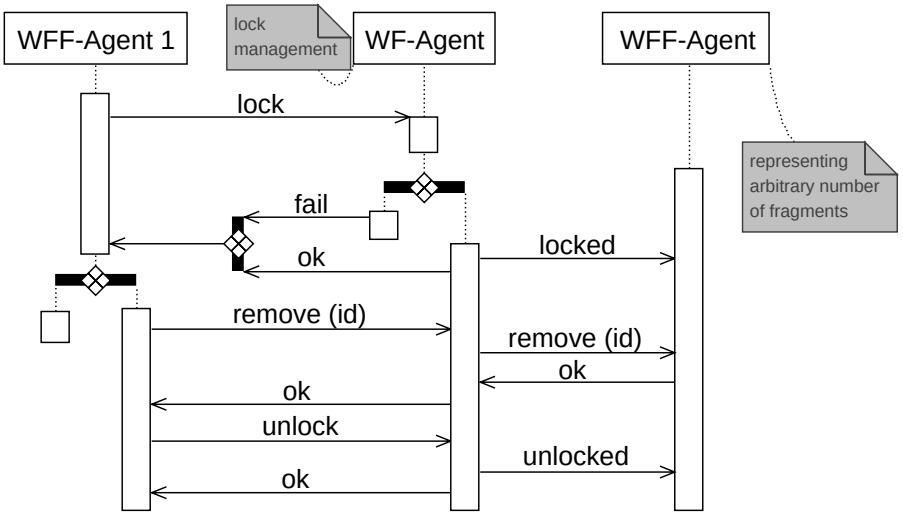

Fig. 6. Synchronization protocol for a border place

The coordination and conflict solving is done by the WF agent, which also manages the lock. The resulting topology has a star shape, so the fragments do not need to know each other. Fig. 6 illustrates this. Further details are described in [24].

Starting and ending workflows or parts of workflows happens through message exchange: a workflow cannot be explicitly stopped, as explained in Section 3.3. The responsibility for an unambiguous termination of a workflow remains with the designer of the workflow itself. After the activation of the `end` transition in the control net the requesting agent is informed about the result of the workflow.

Load Distribution and Redundancy. An agent architecture according to FIPA is useful to implement redundancy by several agents that provide the same service. They can reside on several agent platforms. The selection mechanism used to choose a service provider realizes the desired effect like load distribution. The agents that use services of other agents must realize their services in an adaptive way to enable this scenario, i.e. they have to search for alternative services autonomously.

In the architecture proposed here, the WFMS agent is the only agent that is central to one WF agent platform. If this agent also should work more than once on one platform, one needs to ensure that they synchronize their state carefully.

Directory Service. Because the components of the WFMS are only loosely coupled, the system needs a directory service for discovery and coupling of components. Entries in a directory service should have a validity time and describe services and their providers using globally unique names, they should be searchable across platforms and they should be reliable: only authorized registration and manipulation allowed, and unambiguous service descriptions and a reliable relation to the service provider ensured. The FIPA Directory Facilitator (DF) meets most of these requirements. The missing security features are not addressed in this paper. Probably some agreement will be made for the security of the FIPA DF service. Another possibility is implementing a proprietary secure workflow directory service, e.g. provided by the WFES agent. In both cases, all participating agents need to use the provided security functions.

With the Capa network connection plug-in Ace (see [25]), agents can search and publish services worldwide, e.g. within the open agent network openNet (see [26]).

6 Conclusion

The main result is to provide a powerful architecture for flexible workflow systems along with an approach to distribute workflows on different locations through fragmentation. By using high-level Petri nets, i.e. reference nets, a precise modelling technique is applied to describe workflows and to generate arbitrary fragments that can be distributed within a set of workflow management systems. Thus different organizations are allowed to cooperate, based on a precise process model. The concept of agents allows for a flexible, dynamic and autonomous configuration of each workflow and platform. Since workflows are encapsulated by agents, these advantages can be transfered. The disadvantage is the higher complexity of the infrastructure. However, this is inherent to the requirements on distributed workflow organization. The more possibilities are provided in a workflow management, the more infrastructure has to be provided. Agents as the basis of workflows have the potential to fulfil all requirements for a certain implementation and usage price. What should be noted here are the increasing requirements with respect to distribution and the resulting true concurrency (which is more complex than the usual interleaving (round-robin) semantics of other modelling formalisms). Here the use of reference nets and a tool set that really supports such concurrency is of high value.

6.1 Outlook

There are several possibilities to extend our work so far. We will integrate Renew, Mulan, Capa, the agent network connection plug-in Ace and the workflow management system with our Web service tools to provide a Collaborative

Integrated Development Environment (CIDE). First steps are realized in a prototype and are published in [27]. The workload to really meet the technological / practical requirements for an interorganizational workflow management system allows only for prototypical evaluation. Due to our tight conceptual integration of Web Services and agents our workflow concepts can easily be merged with Web Service environments.

References

1. Jennings, N.R.: On agent-based software engineering. Artificial Intelligence **117**(2) (2000) 277–296
2. Purvis, M., Savarimuthu, B.T.R., Purvis, M.K.: Evaluation of a multi-agent based workflow management system modelled using coloured petri nets. In Barley, M., Kasabov, N.K., eds.: PRIMA. Volume 3371 of LNCS., Springer (2004) 206–216
3. Blake, M.B., Gomaa, H.: Object-oriented modelling approaches to agent-based workflow services. In de Lucena, C.J.P., Garcia, A.F., Romanovsky, A.B., Castro, J., Alencar, P.S.C., eds.: SELMAS. Volume 2940 of LNCS., Springer (2003) 111–128
4. Reese, C., Offermann, S., Moldt, D.: Architektur für verteilte, agentenbasierte Workflows. In Schoop, M., Huemer, C., Rebstock, M., Bichler, M., eds.: Service-oriented Electronic Commerce in the context of the Multikonferenz Wirtschaftsinformatik 2006 (MKWI 2006). Volume P-80 of Lecture Notes in Informatics (LNI) - Proceedings., Bonn, Gesellschaft für Informatik, Köllen Druck+Verlag GmbH (2006) 73–87
5. Aalst, W.v.d.: Inheritance of interorganizational workflows to enable business-to-business. Electronic Commerce Research **2**(3) (2002) 195–231
6. Workflow Management Coalition: WfMC workflow reference model. URL `http://www.wfmc.org/standards/model.htm` (2005)
7. Kummer, O.: Introduction to Petri nets and reference nets. Sozionik Aktuell **1** (2001) 1–9 ISSN 1617-2477.
8. Kummer, O.: Referenznetze. Logos, Berlin (2002)
9. Jensen, K.: Coloured Petri Nets: Volume 1; Basic Concepts, Analysis Methods and Practical Use. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin (1992)
10. Valk, R.: Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In Desel, J., ed.: 19th ICATPN. Number 1420 in LNCS, Berlin, Springer (1998)
11. Christensen, S., Hansen, N.D.: Coloured Petri Nets Extended with Channels for Synchronous communication. Technical Report DAIMI PB–390, Computer Science Department, Aarhus University (1992)
12. RENEW: The reference net workshop homepage. (URL `http://www.renew.de/`)
13. Aalst, W.v.d.: Verification of workflow nets. In Azéma, P., Balbo, G., eds.: Application and Theory of Petri Nets 1997. Number 1248 in LNCS, Berlin, Springer (1997) 407–426
14. Aalst, W.v.d., Hofstede, A.t., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and Parallel Databases **14**(3) (2003) 5–51
15. Jacob, T.: Implementierung einer sicheren und rollenbasierten Workflowmanagement-Komponente für ein Petrinetzwerkzeug. Diploma thesis, University of Hamburg (2002)

16. Aalst, W.v.d., Moldt, D., Valk, R., Wienberg, F.: Enacting Interorganizational Workflows Using Nets in Nets. In: Proceedings of the 1999 Workflow Management Conference. Volume 70., University of Münster (1999) 117–136
17. Jacob, T., Kummer, O., Moldt, D.: Persistent Petri Net Execution. Petri Net Newsletter **61** (2001) 18–26
18. Duvigneau, M., Moldt, D., Rölke, H.: Concurrent architecture for a multi-agent platform. In Giunchiglia, F., Odell, J., Weiß, G., eds.: AOSE 2002, Revised Papers and Invited Contributions. Volume 2585 of LNCS., Berlin, Springer (2003)
19. Foundation for Intelligent Physical Agents: FIPA Agent Management Specification. (2004) `http://www.fipa.org/specs/fipa00023/`.
20. Cabac, L., Moldt, D., Rölke, H.: A Proposal for Structuring Petri Net-Based Agent Interaction Protocols. In van der Aalst, W., Best, E., eds.: 24nd ICATPN 2003, Eindhoven, NL. Volume 2679., Berlin, Springer (2003) 102 – 120
21. Workflow Management Coalition (WfMC). URL `http://www.wfmc.de/` (2005)
22. Lehmann, K., Markwardt, V.: Proposal of an Agent-based System for Distributed Software Development. In Moldt, D., ed.: Proc of MOCA 2004, Aarhus, Denmark (2004) 65–70
23. Bauer, T., Reichert, M., Dadam, P.: Intra-subnet load balancing in distributed workflow management systems. Int. J. Cooperative Inf. Syst. **12**(3) (2003) 295–324
24. Carl, T.: Entwicklung eines agentenbasierten verteilten Workflow-Management-Systems mit Referenznetzen. Diploma thesis, University of Hamburg (2004)
25. Reese, C., Duvigneau, M., Köhler, M., Moldt, D., Rölke, H.: Agent–based settler game. In: Agentcities Agent Technology Competition, Barcelona, Spain. (2003)
26. OpenNet: Project website. `http://www.x-opennet.org/` (2004)
27. Markwardt, K., Moldt, D., Offermann, S., Reese, C.: Using multi-agent systems for change management processes in the context of distributed software development processes. In Sadiq, S., Reichert, M., Schulz, K., eds.: The 1st International Workshop on Technologies for Collaborative Business Processes (TCoB-2006) in the context of the 8th International Conference on Enterprise Information Systems (ICEIS-2006). (2006) accepted contribution.

An Agent-Based Meta-level Architecture for Strategic Reasoning in Naval Planning

Mark Hoogendoorn[1], Catholijn M. Jonker[3], Peter-Paul van Maanen[1,2], and Jan Treur[1]

[1] Vrije Universiteit Amsterdam, Dept. of Artificial Intelligence,
De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands
{mhoogen, treur}@cs.vu.nl
http://www.cs.vu.nl/~{mhoogen, treur}
[2] TNO Human Factors, Dept. of Information Processing,
P.O. Box 23, 3769 ZG Soesterberg, The Netherlands
peter-paul.vanmaanen@tno.nl
http://www.cs.vu.nl/~pp
[3] Radboud University Nijmegen, Nijmegen Institute for Cognition and Information,
Montessorilaan 3, 6525 HR Nijmegen, The Netherlands
c.jonker@nici.ru.nl
http://www.nici.ru.nl/~catholj

Abstract. The management of naval organizations aims at the maximization of mission success by means of monitoring, planning and strategic reasoning. This paper presents an agent-based meta-level architecture for the improvement of automated strategic reasoning in naval planning. The architecture is instantiated with decision knowledge acquired from naval domain experts and is formed into an executable agent-based model, which is used to perform a number of simulation runs. To evaluate the simulation results, relevant properties for the planning decision are identified and formalized. These important properties are validated for the simulation traces.

Keywords: Meta-reasoning, planning, intelligent agent systems.

1 Introduction

The management of naval organizations aims at the maximization of mission success by means of monitoring, planning and strategic reasoning. In this domain, strategic reasoning more in particular helps in determining in resource-bounded situations if a go or no go should be given to, or to shift attention to, a certain evaluation of possible plans after an incident. An incident is an unexpected event that results in an unmeant chain of events if left alone. Strategic reasoning in a planning context can occur both in plan generation strategies (cf. [1]) and plan selection strategies.

The above context gives rise to two important questions. Firstly, what possible plans are first to be considered? And secondly, what criteria are important for selecting a certain plan for execution? In resource-bounded situations, first generated plans should have a high probability to result in a mission success, and the criteria to determine this should be as sound as possible.

In this paper, a generic agent-based meta-level architecture (cf. [2]) is presented for planning, extended with a strategic reasoning level. Besides the introduction of an

M. Kolp et al. (Eds.): AOIS 2005, LNAI 3529, pp. 216 – 230, 2006.

agent-based meta-level architecture, expert knowledge is used in this paper to formally specify executable properties for each of the components of the agent architecture. In contrast to other approaches, this can be done on a conceptual level. These properties can be used for simulation and facilitate formal validation by means of verification of the simulation results. Furthermore, specific evaluation criteria for plans are introduced and formalized in this paper that are appropriate for the naval domain. These include mission success, troop morale and safety of the ships and troops.

The agent architecture and its components are described in Section 2. Section 3 presents the method used to formalize the architecture. Section 4 presents each of the individual components on a more detailed level and instantiates them with knowledge from the naval domain. Section 5 describes a case study and discusses simulation results. In Section 6, a number of properties of the model's behaviour are identified and formalized. A formal tool TTL Checker is used to check the validity of these properties in the simulated traces. Section 7 is a discussion.

2 An Agent-Based Meta-level Architecture for Naval Planning

The agent-based architecture has been specified using the DESIRE framework [3]. For a comparison of DESIRE with other agent-based modelling techniques, such as GAIA, ADEPT, and MetateM, see [4, 5]. The top-level of the system is shown in Fig. 1 and consists of the ExternalWorld and the Agent. Note that this architecture concerns a multi-agent system. This paper however only describes the architecture of a single agent. The ExternalWorld generates observations which are forwarded to the Agent, and executes the actions that have been determined by the Agent. The composition of the Agent is based on the generic agent model described in [6] of which two components are used: WorldInteractionManagement and OwnProcessControl, as shown in Fig. 2. WorldInteractionManagement takes care of monitoring the observations that are received from the ExternalWorld. In the case when these observations are consistent with the current plan, the actions which are specified in the plan are executed by means of

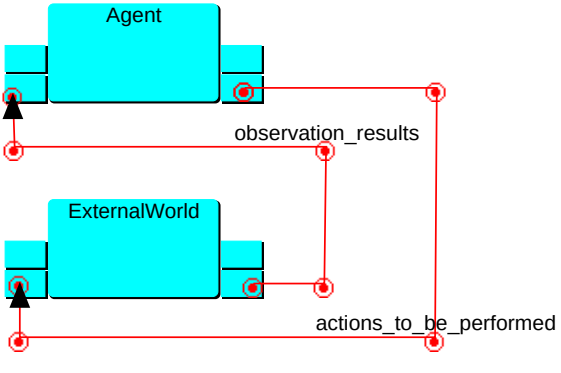

Fig. 1. Top-level architecture

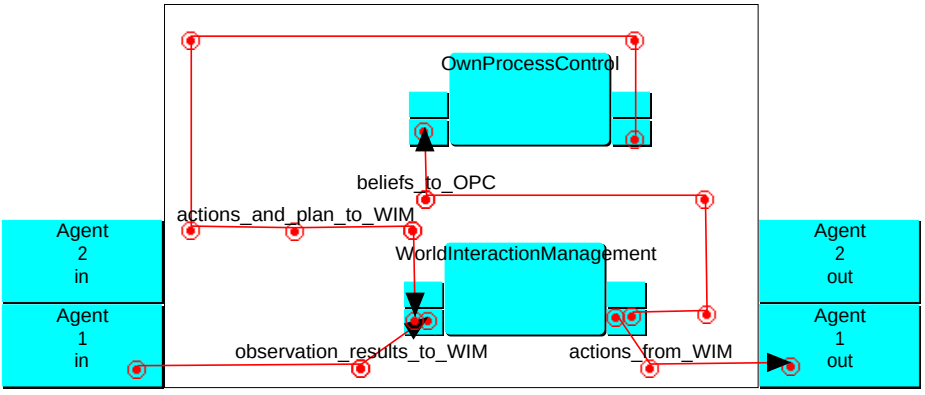

Fig. 2. Agent architecture

forwarding them to the top-level. Otherwise, evaluation information is generated and forwarded to the OwnProcessControl component. Once OwnProcessControl receives such an evaluation, it determines whether the current plan needs to be changed and, in the case that it does, forwards this new plan to WorldInteractionManagement.

WorldInteractionManagement can be decomposed into two components, namely Monitoring and PlanExecution which take care of the tasks as previously presented (i.e. monitoring the observations and executing the plan). For the sake of brevity, the figure regarding these components has been omitted.

OwnProcessControl can also be decomposed, which is shown in Fig. 3. Three components are present within OwnProcessControl: StrategyDetermination, PlanGeneration, and PlanSelection. The PlanGeneration component determines which plans are suitable, given the evaluation information received in the form of beliefs from WorldInteractionManagement, and the conditional rules given by StrategyDetermination. The candidate plans are forwarded to PlanSelection where the most appropriate plan is selected. In the case that no plan can be selected in PlanSelection, this information is forwarded to the StrategyDetermination component. StrategyDetermination reasons on a

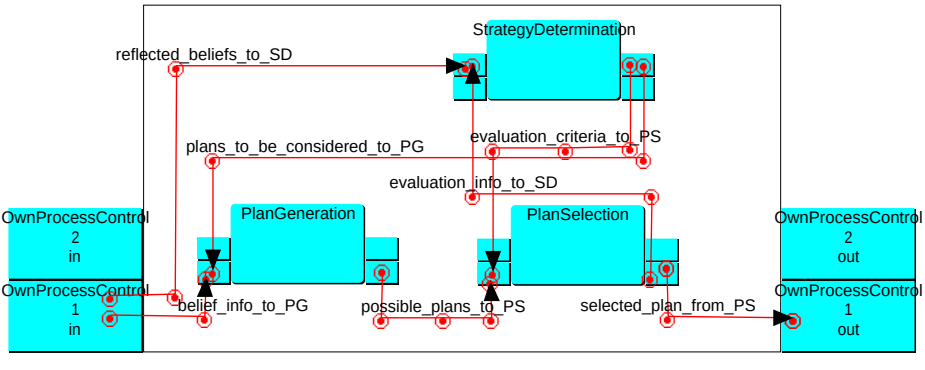

Fig. 3. Components within OwnProcessControl

meta-level (the input is located on a higher level as well as the output as shown in Fig. 3), getting input by translating beliefs into reflected beliefs and by means of receiving the status of the plan selection process from PlanSelection. The component has the possibility to generate more conditional rules and pass them to PlanGeneration, or can change the evaluation criteria in PlanSelection by forwarding these criteria.

The model has some similarities with the model presented in [7]. A major difference is that an additional meta-level is present in the architecture presented here for the StrategyDetermination component. The advantage of having such an additional level is that the reasoning process will be more efficient, as the initial number of options are limited but are required to be the most straightforward ones.

3 Formalization Method

In this section, the method used for the formalization of the model presented in Section 2 is explained in more detail. To formally specify dynamic properties that are essential in naval strategic planning processes and therefore essential for the components within the agent, an expressive language is needed. To this end, the Temporal Trace Language (TTL) is used as a tool cf. [8]. In this section of the paper, both an informal and formal representation of the properties are given.

A state ontology is a specification (in order-sorted logic) of a vocabulary. A state for ontology Ont is an assignment of truth-values {true, false} to the set At(Ont) of ground atoms expressed in terms of Ont. The *set of all possible states* for state ontology Ont is denoted by STATES(Ont). The set of *state properties* STATPROP(Ont) for state ontology Ont is the set of all propositions over ground atoms from At(Ont). A fixed *timeframe* T is assumed that is linearly ordered. A *trace* or *trajectory* γ over a state ontology Ont and timeframe T is a mapping $\gamma : T \rightarrow STATES(Ont)$, i.e. a sequence of states γ_t ($t \in T$) in STATES(Ont). The set of all traces over state ontology Ont is denoted by TRACES(Ont). Depending on the application, the timeframe T may be dense (e.g. the real numbers), or discrete (e.g. the set of integers or natural numbers or a finite initial segment of the natural numbers) or any other form, as long as it has a linear ordering. The set of *dynamic properties* DYNPROP(Σ) is the set of temporal statements that can be formulated with respect to traces based on the state ontology Ont in the following manner.

Given a trace γ over state ontology Ont, the input state of a component c within the agent (e.g. PlanGeneration, or PlanSelection) at time point t is denoted by state(γ, t, input(c)). Analogously state(γ, t, output(c)) and state(γ, t, internal(c)) denote the output state, internal state and external world state.

These states can be related to state properties via the formally defined satisfaction relation |=, comparable to the Holds-predicate in the Situation Calculus: state(γ, t, output(c)) |= p denotes that state property p holds in trace γ at time t in the output state of agent-component c. Based on these statements, dynamic properties can be formulated in a formal manner in a sorted first-order predicate logic with sorts T for time points, Traces for traces and F for state formulae, using quantifiers over time and the usual first-order logical connectives such as $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\forall$, $\exists$. In trace descriptions, notations such as state(γ, t, output(c))|= p are shortened to output(c)|p.

To model direct temporal dependencies between two state properties, the simpler *leads to* format is used. This is an executable format defined as follows. Let α and β be state properties of the form 'conjunction of literals' (where a literal is an atom or the negation of an atom), and e, f, g, h are non-negative real numbers. In the *leads to* language $\alpha \rightarrow\!\!\!\!\rightarrow_{e, f, g, h} \beta$, means:

if state property α holds for a certain time interval with duration g, then after some delay (between e and f) state property β will hold for a certain time interval of length h.

For a precise definition of the *leads to* format in terms of the language TTL, see [9]. A specification of dynamic properties in *leads to* format has as advantages that it is executable and that it can easily be depicted graphically.

4 Component Specification for Naval Planning

This section introduces each of the components within the strategic planning process in more detail. The components presented in this section are only those parts of OwnProcessControl within the agent since they are most relevant for the planning process. A partial specification of executable properties in semi-formal format is also presented for each of these components. The properties introduced in this section are generic for naval (re)planning and can easily be instantiated with mission-specific knowledge. All of these properties are the result of interviews with officers of the Royal Netherlands Navy.

4.1 Plan Generation

The rules for generation of a plan can be stated very generally as the knowledge about plans. Conditions for those plans are stored in the StrategyDetermination component, which is treated later. Basically, in this domain, the component contains one rule:

if belief(S:oSITUATION, pos)
and conditionally_allowed(S:SITUATION, P:PLAN)
then candidate_plan(P:PLAN)

stating that in the case that Monitoring evaluated the current situation as being situation S and the PlanGeneration has received an input that situation S allows for plan P, then it is a candidate plan. This information is passed to the PlanSelection component.

4.2 Plan Selection

Plan selection is the next step in the process and for this domain there are three important criteria that determine whether a plan is appropriate or not: (1) Mission success, (2) safety and (3) a fleet morale criterion. In this scenario, it is assumed that a weighted sum can be calculated and used in order to make a decision between candidate plans. The exact weight of each criterion is determined by the StrategyDetermination component. The value for the criteria can be derived from observations in the world and, for example, a weighted sum can be taken over time. To obtain the observations, for each candidate plan the consequent events of the plan

are determined and formed into an observation. Thereafter, the consequences of these observations for the criteria can be determined. In the examples shown below, the bridge between changes of the criteria after an observation and the overall value of the criteria are not shown in a formal form for the sake of brevity.

Mission Success. An important criterion is of course the mission success. Within this criterion, the objective of the mission plays a central role. In the case that a certain decision needs to be made, the influence this decision has for the mission success needs to be determined. The criterion involves taking into account several factors. First of all, the probability that the deadline is reachable. Besides that, the probability that the mission succeeds with a specific fleet configuration. The value of the mission success probability is a real number between 0 and 1. A naval domain expert has labelled certain events with an impact value on mission success. This can entail a positive effect or a negative effect. The mission starts with an initial value for success, taking into consideration the assignment and the enemy. In the case that the situation changes this can lead to a change of the success value. An example of an observation with a negative influence is shown below.

```
if    current_success_value(S:REAL)
 and  belief(ship_left_behind, pos)
then  new_succes_value(S:REAL * 0.8)
```

Safety. Safety is an important criterion as well. When a ship loses propulsion, the probability of survival decreases dramatically if left alone. Basically, the probability of survival depends on three factors: (1) the speed with which the task group is sailing, (2) the configuration of own ships, which includes the amount and type of ships, and their relative positions and (3) the threat caused by the enemy, the kind of ships the enemy has, the probability of them attacking the task group etc.

The safety value influences the evaluation value of possible plans. The duration of a certain safety value determines its weight in the average risk value, so a weighted sum based on time duration is taken. The value during a certain period in time is again derived by means of an initial safety value and events in the external world causing the safety value to increase or decrease. An example rule:

```
if    current_safety_value(S:REAL)
 and  belief(speed_change_from_to(full, slow), pos)
then  new_safety_value(0.5 * S:REAL)
```

Fleet morale. The morale of the men on board of the ships is also important as a criterion. Morale is important in the considerations as troops with a good morale are much more likely to win compared to those whose morale is low. Troop morale is represented by a real number with a value between 0 and 1 and is determined by events in the world observed by the men. Basically, the men start with a certain morale value and observations of events in the world can cause the level to go up or down, similar to the mission success criterion. One of the negative experiences for morale is the observation of being left behind without protection or seeing others solely left behind:

```
if    current_morale_value(M:REAL)
 and  belief(ship_left_behind, pos)
then  new_morale_value(M:REAL * 0.2)
```

An observation increasing the morale is that of sinking an enemy ship:

```
if      current_morale_value(M:REAL)
and   belief(enemy_ship_eliminated, pos)
and   min(1, M:REAL * 1.6, MIN:REAL)
then  new_morale_value(MIN:REAL)
```

4.3 Strategy Determination

The StrategyDetermination component within the model has two functions: first of all, it determines the conditional plans that are to be used given the current state, secondly, it provides a strategy for the selection of these plans.

In general, naval plans are generated according to a preferred plan library or in exceptional cases outside of this preferred plan library. The StrategyDetermination component within the model determines which plans are to be used and thereafter forwards these plans to the PlanGeneration component. The StrategyDetermination component determines one of three modes of operation on which conditional rules are to be used in this situation:

1. **Limited action demand.** This mode is used as an initial setting and is a subset of the preferred plan library. It includes the more common actions within the preferred plan library;
2. **Full preferred plan library.** Generate all conditional rules that are allowed according to the preferred plan library. This mode is taken when the limited action mode did not provide a satisfactory solution;
3. **Exceptional action demand.** This strategy is used in exceptional cases, and only in case the two other modes did not result in an appropriate candidate plan.

Note that the plans within the first mode of operation occur much more frequently than the ones in the second mode. A similar relation holds between the second and the third mode of operation. The idea is that the result of this frequency difference while using a strategy determination component is a significant improvement of the reasoning process.

Next to determining which plans should be evaluated, the StrategyDetermination component also determines *how* these plans should be evaluated. In Section 4.3, it was stated that the plan selection depends on mission success, safety and fleet morale. All three factors determine the overall evaluation of a plan to a certain degree. Plans can be evaluated by means of an evaluation formula, which is described by a weighted sum. Differences in weights determine differences in plan evaluation strategy. The plan evaluation formula is as follows (in short):

$$\text{evaluation\_value(P:PLAN)} = \alpha * \text{mission\_success\_value(P:PLAN)} + \beta * \text{safety\_value(P:PLAN)} + \gamma * \text{fleet\_morale\_value(P:PLAN)}$$

where all values and degrees are in the interval [0,1] and $+ + = 1$. The degrees depend on the type of mission and the current state of the process. For instance, if a mission is supposed to be executed safely at all cost or the situation shows that already many ships have been lost, the degree should be relatively high.

In this case the following rules hold:

```
if      problem_type(mission_success_important)
 and    problem_type(safety_important)
 and    problem_type(fleet_morale_important)
 and    candidate_plan(P:PLAN)
 and    mission_success_value(P:PLAN, R1:REAL)
 and    safety_value(P:PLAN, R2:REAL)
 and     fleet_morale_value(P:PLAN, R3:REAL)
then    evaluation_value(no_propulsion(ship), 0.33 * R1:REAL + 0.33 * R2:REAL + 0.33 *R3:REAL)
```

In case two criteria are most important the following rule holds:

```
if      problem_type(mission_success_important)
 and    problem_type(safety_important)
 and    not problem_type(fleet_morale_important)
 and    candidate_plan(P:PLAN)
 and    mission_success_value(P:PLAN, R1:REAL)
 and    safety_value(P:PLAN, R2:REAL)
 and     fleet_morale_value(P:PLAN, R3:REAL)
then    evaluation_value(no_propulsion(ship), 0.45 * R1:REAL + 0.45 * R2:REAL + 0.1 *R3:REAL)
```

This holds for each of the problem type combinations where two criteria are important: A weight of 0.45 in case the criterion is important for the problem type and 0.1 otherwise. Finally, only one criterion can be important:

```
if      problem_type(mission_success_important)
 and     not problem_type(safety_important)
 and    not problem_type(fleet_morale_important)
 and    candidate_plan(P:PLAN)
 and    mission_success_value(P:PLAN, R1:REAL)
 and    safety_value(P:PLAN, R2:REAL)
 and     fleet_morale_value(P:PLAN, R3:REAL)
then    evaluation_value(no_propulsion(ship), 0.6 * R1:REAL + 0.2 * R2:REAL + 0.2 *R3:REAL)
```

The plan generation modes and plan selection degrees presented above can be specified by formal rules which have been omitted for the sake of brevity.

5 Case-Study: Total Steam Failure

This section presents a case study which has been formalized using the agent-based model presented in Section 2 and 4. This case study is again based upon interviews with expert navy officers of the Royal Netherlands Navy. The formalization of this process follows the methodology presented in Section 3.

5.1 Scenario Description

The scenario used as an example is the first phase within a *total steam failure* scenario. A fleet consisting of 6 frigates (denoted by F1 – F6) and 6 helicopters (denoted by H1 – H6) are protecting a specific area called Zulu Zulu (denoted by ZZ). For optimal protection of valuable assets that need to be transported to a certain location and need to arrive before a certain deadline, the ships carrying these assets

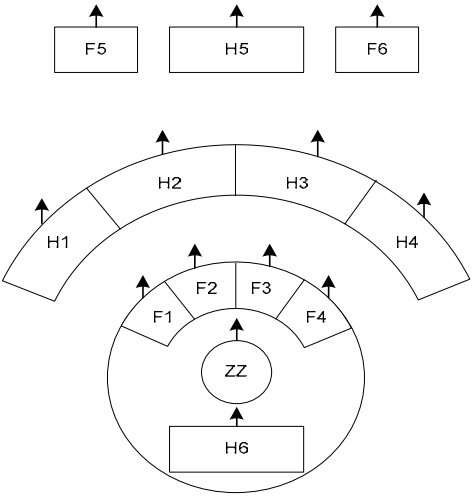

Fig. 4. Scenario for meta-reasoning

are located in ZZ. These ships should always maintain their position in ZZ to guarantee optimal protection. The formation at time T0 is shown in Fig. 4. On that same time-point the following incident occurs: an amphibious transport ship that is part of ZZ loses its propulsion and cannot start the engines within a few minutes. When a mission is assigned to a commander of the task group (CTG), he receives a preferred plan library from the higher echelon. This library gives an exhaustive list of situations and plans that are allowed to be executed within that situation. Therefore the CTG has to make a decision: what to do with the ship and the rest of the fleet. In the situation occurring in the example scenario, the preferred plan library consists of four plans:

1. **Continue sailing.** Leave the ship behind. The safety of the main fleet will therefore be maximal, althiough the risk for the ship is high. The morale of all the men within the fleet will drop.
2. **Stop the entire fleet.** Stopping the fleet ensures that the ship is not left behind and lost, although the risks for the other ships increase rapidly as an attack is more likely to be successful when not moving.
3. **Return home without the ship.** Rescue the majority of the men from the ship, return home, but leave a minimal crew on the ship that will still be able to fix the ship. The ship will remain in danger until it is repaired and the mission is surely not going to succeed. The morale of the men will drop to a minimal level. This option is purely hypothetical according to the experts.
4. **Form a screen around the ship.** This option means that part of the screen of the main fleet is allocated to form a screen around the ship. Therefore the ship is protected and the risks for the rest of the fleet stay acceptable.

Option 4 involves a lot more organizational change compared to the other options and is therefore considered after the first three options. The CTG decides to form a screen around the ship.

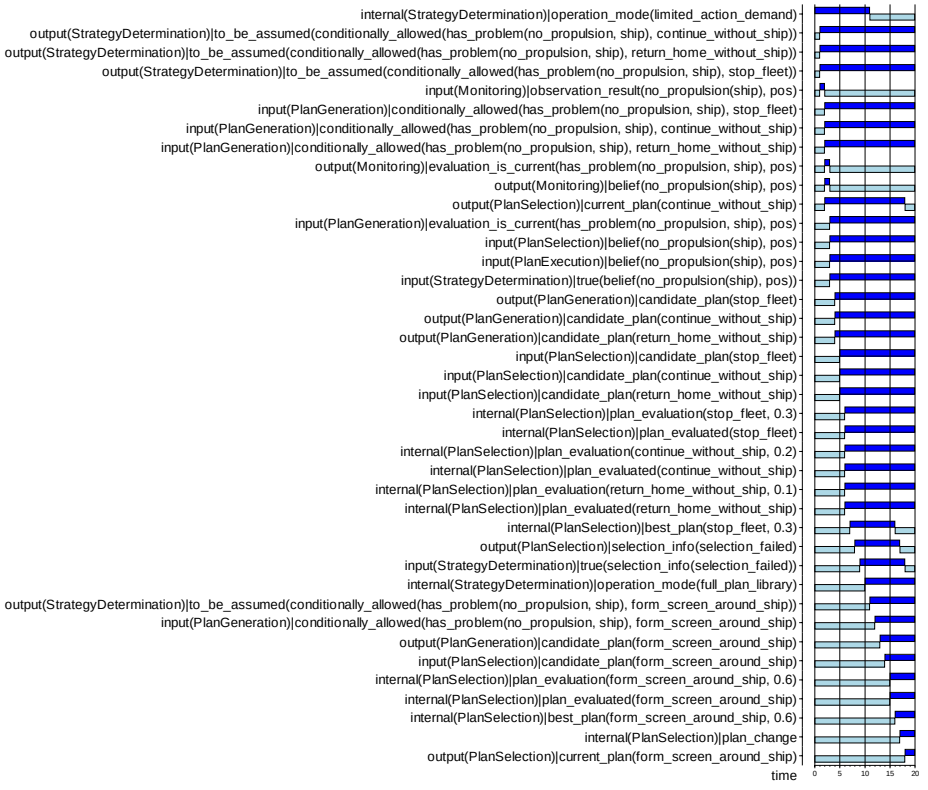

Fig. 5. Trace of the total steam failure simulation

5.2 Simulation Results

The most interesting results of the simulation using the architecture and properties described in Section 2 and 4, and instantiated with the case-study specific knowledge from Section 5.1, are shown in Fig. 5. The trace, a temporal description of chains of events, describes the decision making process of the agent that plays the role of Commander Task Group (CTG). The atoms on the left side denote the information between and within the components of the agents. To keep the figure clear, only the atoms of the components on the lowest level of the agent architecture are shown. The right side of the figure shows when these atoms are true. In the case of a black box the atom is true during that period, in other cases the atom is false (closed world assumption). The atoms used are according to the model presented in Section 2. For example, internal(PlanGeneration) denotes that the atom is internal within the PlanGeneration component. More specifically, the trace shows that at time-point 1 the Monitoring component receives an input that the ship has no propulsion

input(Monitoring)|observation_result(no_propulsion(ship), pos)

The current plan is to continue without the ship, as the fleet continues to sail without any further instructions:

```
output(PlanSelection)|current_plan(continue_without_ship)
```

As the StrategyDetermination component always outputs the options currently available for all sorts of situations (in this case only a problem with the propulsion of a ship), it continuously outputs the conditionally allowed information in the limited action mode, for example:

```
output(StrategyDetermination)|to_be_assumed(
        conditionally_allowed(has_problem(no_propulsion, ship),continue_without_ship))
```

The information becomes an input through downward reflection, a translation from a meta-level to a lower meta-level:

```
input(PlanGeneration)|conditionally_allowed(
                        has_problem(no_propulsion, ship), continue_without_ship)
```

The Monitoring component forwards the information about the observation to the components on the same level as beliefs. The StrategyDetermination component also receives this information but, instead of a belief, it arrives as a reflected belief through upward reflection that is a translation of information at a meta-level to a higher meta-level:

```
input(StrategyDetermination)| true(belief(no_propulsion(ship), pos))
```

Besides deriving the beliefs on the observations, the Monitoring component also evaluates the situation and passes this as evaluation information to the PlanGenerator.

```
input(PlanGenerator)|evaluation(has_problem(no_propulsion, ship), pos)
```

This information acts as a basis for the PlanGenerator to generate candidate plans, which are sent to the PlanSelection, for example.

```
input(PlanSelection)|candidate_plan(continue_without_ship)
```

Internally the PlanSelection component determines the evaluation value of the different plans, compares them and derives the best plan out of the candidate plans:

```
internal(PlanSelection)|best_plan(stop_fleet, 0.3)
```

This value is below the threshold evaluation value and therefore the PlanSelection component informs the StrategyDetermination component that no plan has been selected:

```
output(PlanSelection)|selection_info(selection_failed)
```

Thereafter the StrategyDetermination component switches to the full preferred plan library and informs PlanGeneration of the new options. PlanGeneration again generates all possible plans and forwards them to PlanSelection. PlanSelection now finds a plan that is evaluated above the threshold and makes that the new current plan.

```
output(PlanSelection)|current_plan(form_screen_around_ship)
```

This plan is forwarded to the PlanExecution and Monitoring components (not shown in the trace) and is executed and monitored.

6 Validation by Verification

After that a formalized trace has been obtained in the previous section, either by formalization of an empirical trace or by means of simulation, in this section it is validated whether the application of the model complies to certain desired properties of this trace. The verification of these properties in the trace is shown below. The properties are independent from the specific scenario and should hold for every scenario for which the agent-based meta-level architecture presented in Sections 2 and 4 is applied. The properties are formalized using Temporal Trace Language as described in Section 3.

P1: Upward reflection. This property states that information generated at the level of the Monitoring and PlanSelection components should always be reflected upwards to the level of the StrategyDetermination component. Of course, since this translation is being performed automatically, this property is rather straightforward but checking such a property can be useful to see whether the system indeed functions correctly. In semi-formal notation, the property is specified as follows:

At any point in time t,
if Monitoring outputs a belief about the world at time t
then at a later point in time t2 StrategyDetermination receives this information through upward reflection
At any point in time t,
if PlanSelection outputs selection info at time t
then at a later point in time t2 StrategyDetermination receives this information through upward reflection.

In formal form, the property is as follows:

$\forall t$ [[$\forall$O:OBS, S:SIGN [state(γ, t, output(Monitoring)) |= belief(O, S)
$\Rightarrow$ $\exists t2 \geq t$ state(γ, t2, input(StrategyDetermination)) |= true(belief(O,S))]]
& [$\forall$SI:SEL_INFO [state(γ, t, output(PlanSelection)) |= selection_info(SI)
$\Rightarrow$ $\exists t2 \geq t$ state(γ, t2, input(StrategyDetermination)) |= true(selection_info(SI))]]]

This property has been automatically checked and thus shown to be satisfied within the trace.

P2: Downward reflection. Property P2 verifies that all information generated by the StrategyDetermination component for a lower meta-level is made available at that level through downward reflection. In formal form:

$\forall$t, S:SITUATION, P:PLAN [state(γ, t, output(StrategyDetermination))
|= to_be_assumed(conditionally_allowed(S, P))
$\Rightarrow \exists t2 \geq t$ state(γ, t2, input(PlanGeneration)) |= conditionally_allowed(S, P)]

This property is also satisfied for the given trace.

P3: Extreme measures. This property states that measures that are not part of the preferred plan library (extreme measures) are only taken in case some other options failed. In formal form:

$\forall t, t2 > t$, S:SITUATION, P1:PLAN, P2:PLAN
[[state(γ, t, output(Monitoring)) |= evaluation(exception(S), pos) & state(γ, t, output(PlanSelection)) |=
current_plan(P1) & state(γ, t2, output(PlanSelection)) |= current_plan(P2) & P1 $\neq$ P2
& $\neg$state(γ, t2, internal(StrategyDetermination)) |= to_be_assumed(preferred_plan(S, P2)]
$\Rightarrow \exists t'$ [$t' \geq t$ & $t' \leq t2$ & state(γ, t', output(PlanSelection)) |= selection_info(selection_failed)]]

The property is satisfied for the given trace.

P4: Plans are changed only if an exception was encountered. Property P4 formally describes that a plan is only changed in case there has been an exception that triggered this change. Formally:

$\forall t, t2 \geq t,\ P{:}PLAN\ [\ [state(\gamma, t, output(PlanSelection)) \models current\_plan(P)\ \&$
$\neg state(\gamma, t2, output(PlanSelection)) \models current\_plan(P)] \Rightarrow \exists t', S{:}SITUATION\ [t' \geq t\ \&\ t' \leq t2\ \&$
$state(\gamma, t', output(Monitoring)) \models evaluation(exception(S), pos)]\ ]$

This property is again satisfied for the given trace.

7 Discussion

This paper presents an agent-based architecture for strategic planning (cf. [1]) for naval domains. The architecture was designed as a meta-level architecture (cf. [2]) with three levels. The interaction between the levels in this paper is modelled by reflection principles (e.g. [10]). The dynamics of the architecture is based on a multi-level trace approach as an extension of that described in [11]. The architecture has been instantiated with naval strategic planning knowledge. The resulting executable model has been used to perform a number of simulation runs. To evaluate the simulation results, desired properties for the planning decision process have been identified, formalized and then validated for the simulation traces. The framework that was presented was illustrated by means of a case study in the navel domain. A more domain independent architecture and its application in other domains will be addressed in future work.

A meta-level architecture for strategic reasoning in another area, namely that of design processes, is described in [12]. This architecture has been used as a source of inspiration for the current architecture for strategic planning. In other architectures, such as in PRS [13], meta-level knowledge is also part of the system, although this knowledge is not explicitly part of the architecture (it is part of the Knowledge Areas) as is the case in the architecture presented in this paper.

Agent models of military decision making have been investigated before. In [14], for example, an agent-based model is presented that mimics the decision process of an experienced military decision maker. Potential decisions are evaluated by checking if they are good for the current goals. A case study of decisions to be made at an amphibian landing mission is used. The outcome of the evaluations of the decisions that can be made in the case-study are compared to the decisions made by real military commanders. The approach presented is different from the approach taken in this paper as a more formal approach is taken here to evaluate the model created. Also the focus in this paper is more on the model of the decision maker itself and not on the correctness of the decisions, which is the case in [14]. The main advantage of the approach taken is that the system is specified and can be simulated on a conceptual level contrary to other approaches. Furthermore, for knowledge intensive domains, such as the naval domain, there is the problem of scalability. This is acknowledged by the authors and suggest further research for different domains and variants. It is possible for instance to add or change the described criteria or apply particular planning algorithms. Finally, this paper addressed resource-bounded situations.

In [15] an overview is presented of models for human behaviour that can be used for simulations. Similar to research done in other agent-based systems using the DESIRE framework [3], future research in simulation and the validation of relevant properties for the resulting simulation traces is expected to give key insight for the implementation of future complex resource-bounded agent-based planning support systems used by commanders on naval platforms.

Acknowledgments

CAMS-Force Vision, a software development company associated with the Royal Netherlands Navy, funded this research and provided domain knowledge. The authors especially want to thank Jaap de Boer (CAMS-Force Vision) for his expert knowledge. Finally, the authors would like to thank the anonymous reviewers for their useful comments.

References

1. Wilkins, D.E.: Domain-independent planning Representation and plan generation. Artificial Intelligence 22 (1984) 269-301
2. Maes, P, Nardi, D. (eds): Meta-level architectures and reflection, Elsevier Science Publishers (1988)
3. Brazier, F.M.T., Dunin Keplicz, B., Jennings, N., Treur, J.: DESIRE: Modelling Multi-Agent Systems in a Compositional Formal Framework. *International Journal of Cooperative Information Systems*, 6 (1997) 67-94
4. Shehory, O., Sturm, A.: Evaluation of modeling techniques for agent-based systems, In: Proceedings of the fifth international conference on Autonomous agents, Montreal, Canada (2001) 624-631
5. Mulder, M, Treur, J., Fisher, M.: Agent Modelling in MetateM and DESIRE. In: M.P. Singh, A.S. Rao, M.J. Wooldridge (eds.), *Intelligent Agents IV, Proc. Fourth International Workshop on Agent Theories, Architectures and Languages, ATAL'97.* Lecture Notes in AI, vol. 1365, Springer Verlag (1998) 193-207
6. Brazier, F.M.T., Jonker, C.M., Treur, J.: Compositional Design and Reuse of a Generic Agent Model. *Applied Artificial Intelligence Journal,* 14 (2000) 491-538
7. Jonker, C.M., Treur, J.: A Compositional Process Control Model and its Application to Biochemical Processes. Applied Artificial Intelligence Journal, 16 (2002) 51-71
8. Jonker, C.M., Treur, J.: Compositional verification of multi-agent systems: a formal analysis of pro-activeness and reactiveness. International. Journal of Cooperative Information Systems, 11 (2002) 51-92
9. Jonker, C.M., Treur, J., Wijngaards, W.C.A.: A Temporal Modelling Environment for Internally Grounded Beliefs, Desires and Intentions. Cognitive Systems Research Journal, 4 (2003) 191-210
10. Bowen, K., Kowalski, R.: Amalgamating language and meta-language in logic programming. In: K. Clark, S. Tarnlund (eds.), Logic programming. Academic Press (1982)
11. Hoek, W. van der, Meyer, J.-J.Ch., Treur, J.: Formal Semantics of Meta-Level Architectures: Temporal Epistemic Reflection. International Journal of Intelligent Systems, 18 (2003) 1293-1318

12. Brazier, F.M.T., Langen, P.H.G. van, Treur, J.: Strategic Knowledge in Design: a Compositional Approach. Knowledge-based Systems, 11 Special Issue on Strategic Knowledge and Concept Formation, K. Hori, ed.) (1998) 405-416
13. Georgeff, M. P., Ingrand, F. F.: Decision-making in an embedded reasoning system. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89), Detroit, MI (1989) 972-978,
14. Sokolowski, J., Enhanced Military Decision Modeling Using a MultiAgent System Approach, In Proceedings of the Twelfth Conference on Behavior Representation in Modeling and Simulation, Scottsdale, AZ., May 12-15 (2003) 179-186
15. Pew, R.W., Mavor, A.S.: Modeling Human and Organizational Behavior, National Academy Press, Washington, D.C. (1999)

Coordination Efficiency in Rational Choice Theory, Norm and Rights Based Multi-agent Systems

Peter Kristoffersson and Eduardo Alonso

Department of Computing, City University, London EC1V 0HB, United Kingdom
Tel.: +44 (0)20 7040 8552; Fax: +44 (0)20 7040 0244
{dn184, eduardo}@soi.city.ac.uk

Abstract. As utility calculus cannot account for an important part of agents' behaviour in Multi-Agent Systems, researchers have progressively adopted a more normative approach. Unfortunately, social laws have turned out to be too restrictive in real-life domains where autonomous agents' activity cannot be completely specified in advance and the complexity of the system is ever changing. The idea of Rights is a halfway concept between anarchic and off-line constrained interaction. Rights improve coordination and facilitate social action in Multi-Agent domains through adjusting the coordination mechanisms to the complexity of the system. So far rights have not been tested or proven experimentally. We are comparing experimentally the three mentioned interaction architectures in the domain of agent-based traffic simulation.

1 Introduction

The Rational Choice Theory (RCT) has been the most influential theory for designing agents in Artificial Intelligence and Distributed Artificial Intelligence. According to this approach to rationality, agents with complete knowledge make their decisions in order to maximize their own utilities. In this non-constrained approach, agents are assumed to be "free". They act of their own accord and are not subject to any set of (social) rules. However fruitful this approach has been, there have been pointed out (e.g. [1]) some drawbacks in RCT:

- In real dynamic domains, agents do not have enough information or time to perform complex, optimal utility calculus.
- The utilitarian approach fails to explain cooperation and social action.

In order to cope with these problems, the MAS community has adopted a more constrained approach to rationality including conventions, norms and/or social laws. It is well-known that agents working under norms do not need to calculate continuously their utilities and, consequently, do not need complete information. Agents are supposed to act in a somehow predetermined way according to the principle of "mutual expectation". Besides, norms imply that the agents respect certain social constraints that deter them from breaking agreements. Unfortunately, research in this field has fallen into two extreme positions:

M. Kolp et al. (Eds.): AOIS 2005, LNAI 3529, pp. 231 – 242, 2006.

1. Shoham and Tennenholtz [2] have studied off-line social laws, with which agents must comply automatically. Here, the agents are assumed to follow rules just because they are designed to do so. Following this line of argumentation, agents are not seen as autonomous any more. Proposals so formulated are thus closer to Distributed Problem Solving than to MAS.

2. Alternatively, conventions [3] have been introduced as rules emerging during repeated encounters in open normative systems. The problem here is that no notion of sanction is considered. Consequently, if the agents have the chance to calculate their utility each time they interact, conventions are continually under consideration. In other words, following a convention is not always a stable strategy.

A further dimension to this is the fact that, in most agent systems, the dynamics and thus complexity of the environment will be ever-changing. It is well known that an RCT architecture will perform well in situations with complete or near-complete knowledge. It is also well known that off-line designed social laws are very efficient in complex systems. Little work has been done on studying how the changing complexity of an environment affects successful coordination. Excelente-Toledo and Jennings [4] are tackling a similar problem by introducing a system where the coordination mechanism is selected by agents at runtime through reasoning about the task at hand and its importance. Although novel and interesting, this approach has a few weaknesses. Firstly, in situations where the dynamics change back and forth between static and highly dynamic, it might not be feasible to use it and, secondly, it requires an evaluation set of each possible scenario and coordination mechanism used in that instance. It seems, therefore, that we need a concept that:

a) Allows agents to reason and make decisions

b) Implies enforcement at the same time and

c) Can adjust itself to varying complexity levels

The idea of "right" has been proposed by Alonso [5] as a coordination mechanism that deals with a) and b). Rights have been further explained and axiomatically represented in [6]. We believe that rights through their automatic adjustment can also solve c). A theoretical comparison of the three methods can be seen in Table 1.

So far it has still not been experimentally proven whether rights work and how well this mechanism performs in real life situations. We will therefore explore and compare experimentally off-line designed rights with off-line designed social laws (focussing on obligations and prohibitions) and the RCT architecture. Due to space constraints, we will not explain RTC or social norms in much detail. The reader is assumed to be familiar with game theoretic and normative approaches to MAS coordination. Neither will we discuss other alternatives (such as bounded rationality etc.) to RCT since Rights are more related to Norms than other solutions. The reminder of the paper is structured as follows. In the second section we present the concept of rights in more detail and what we gain by introducing them. Section 3 present the

system in which we test the architectures while Section 4 describes how these architectures were implemented. Section 5 defines the experiment parameters while in Sections 6 and 7 we show the results and analyse them. We finish with some conclusions and suggestions for further research.

Table 1. Theoretical comparison of Norms, Rights and RCT

	RCT	Norms	Rights
Complexity	High	Low	Low/Medium
Efficiency	Low	High	High
Stability	High	High	High
Flexibility	High	Low	High

2 Rights

Roughly stated, a right is considered as a set of restrictions on the agents' activities that allows them enough freedom, but at the same time constrains them. Not surprisingly, some authors (e.g. [7]) have expressed the same idea from a RCT perspective, by introducing some constraints in the set of strategies available to the agents. In so doing, agents are free to converge on "stable social laws" (qualitative equilibrium). However interesting this approach may be, it presents a serious handicap: to make sure that the agents choose a stable and efficient strategy, the designer decides beforehand which strategies should be eliminated. The designer, therefore, manipulates the process and creates an "illusion of freedom".

Generally speaking, if an agent has the right to execute a set of actions then (a) he/she is permitted to perform it (under certain constraints or obligations), (b) the rest of the group is not allowed to execute any action inhibiting the agent from exercising his right, and (c) the group is obliged to prevent this inhibitory action.

Rights can be modelled as norms but to do so is very difficult. A rights-based system can be seen as a normative system at the instance the decision is being made. The difference is that not all agents will have to obey the norms and that every agent will have a different set of norms in the situation. The set of norms that governs each agent will also be different from one instance to another.

The idea of using rights is worthy of consideration because it makes it easier to have agents coordinated. This has already been described and showed qualitatively by Alonso [6]. As it has been repeatedly pointed out (e.g. [8, 9, 10]), coordination is mainly concerned with complexity, efficiency, stability and flexibility. Rights aid all of these. Coordination based on rights is more efficient. The more complex a situation is, the more will the rights be used thus coordinating different agents. In situations where coordination is unnecessary, the rights will not be executed. This flexibility means that a rights-based coordination mechanism adjust itself to the situation.

For a more comprehensive description and a formal characterization of rights using the language L and the axiomatic proof, the reader is referred to Alonso's [6] work.

2.1 Evaluating Gains

To prove experimentally that rights give good and efficient coordination in systems with different complexity levels, we have decided to create traffic MAS simulation and to test the three mentioned coordination mechanisms in this environment. We have set up two different experiment sets where we are interested in testing the stability and efficiency of the system with regards to agent survival rate and average speed. In the first set, we have an experiment with lanes and no junctions and agent architectures based on each other to ensure that it's not the programming that decides the results. In the second setup, we are looking at junctions. This time the architectures are not based on each other. Even though we will be comparing the outcomes of the three mechanisms, it is important to understand that the results in themselves can always be challenged. Therefore, even though we are evaluating the results, we are more interested in the result patterns rather than the results themselves. The reason for this is that it is very difficult if not impossible to evaluate the mechanisms against each other. There is always a chance that one could design a better architecture that could outperform the others. If we however look at it as finding patterns in behaviour of the coordinated system, we will gain a better understanding of the coordination methods and how they function.

3 Experimental Environment

The reason for using traffic simulation is that this domain is intuitively easy to understand. The created system is based on a microscopic traffic simulation system developed by Tom Fotherby in Java. Our redesign changed most of the internal working of the system with the exception of the time engine, graphics and road design ability. The agent architecture, information provided by the system to the agents and users, data saving, statistics and interaction between the agents (crashes) have been created by us. The internal engine of the system is based on two main methods, a "pretick" and a "tick" in each agent. The system alternates invoking the "pretick" and the "tick" methods in all registered agents. Firstly, all "pretick" methods are run, after which all the "ticks" are run, this going in a loop. This allows the agent to firstly calculate what to do next (in the "pretick") without any risk that the environment will change before the actions can be implemented. Then, in the "tick" all these actions can be implemented so that they happen simultaneously from the agents' perspective. The time measurement in the system is done through steps where one time step is defined as one loop of "preticks" and "ticks".

The system allows agents to perceive their environment forward, backwards and to the sides back and forth. It gives full information about the distance to other agents as long as the other agent is on the same stretch of the road. It also gives their speeds and direction. In a lane, the agent can only see one agent ahead meaning that if we have three agent-cars driving in a row in front of us, we will only see the closest one. In the junction, it can see the three closes agents in front, 3 behind and 3 on each side. The system also generates traffic lights (green, orange and red), which are visible to the agents. The agents can change their speed and position on the road (lane) and direction in a junction in order to go past obstacles. Each car's maximum speed is set randomly with a minimum of 44 and a maximum of 82. The system permits the definition of the

rate of new incoming agents, where new agents enter (are created) the system every N time steps (one car every N time units => 1/N) at the spot where the lane touches the border of the simulation window and are removed from the system when they crash (after 10 time units) or when they reach the end of the lane. The entry per time unit is connected to each lane (so two lanes in the same direction will have 2* 1 entries every N time units). Every car that crashes will be immovable for 10 time steps after which it will disappear.

4 Agent Architecture

In our experiments, the agent plays the role of a car that wants to survive (not crash) the trip and get through the system as fast as possible. The agents are homogenous. The main goal for the agents is obviously survival. In order to ensure that it was the architecture and not the coding that created better performance in the first experimental setup, the normative architecture is an enhanced free rider architecture and the rights architecture is an enhanced normative architecture. In the second setup, the agents' decision systems are not based on each other anymore. The aim is to show that it's not an improvement of the predecessor that creates the results (as one could argue in scenario one) but rather the architecture itself.

4.1 RCT (Free Rider) Architecture

The free rider (RCT) architecture is a simple deliberative architecture. It allows the choice of the best action for any given situation by evaluating which would allow the agent to perform best (drive faster and not crash) by assigning them numbers between 0 and 4 depending on the suitability of the action. At each time step, the agents are re-evaluating their choices. The agent can only perform one action at a time. The actions are arranged in a hierarchy. The possible actions are (according to their hierarchy): accelerate, do nothing, switch to left lane (turn left in the junction), switch to right lane (turn right in the junction), decelerate. This architecture was selected as it is simple to implement, easy to understand, easy to extend with new choices and allows prioritization between actions when two actions have the same utility figure. The free agents are using this to decide what to do next from their own selfish perspective and are allowed to do whatever they want. They will drive in the wrong lane and through a red light in the junction if it suits their goals. They are only concerned with their own safety.

In summary, for each time step the agent will:
Evaluate all possible actions and assign them utility values
Discover the highest obtained utility value for this time step
Perform the action with highest value and hierarchy

4.2 Normative Architecture

In the first experimental setup, normative agents use the selfish agent architecture with an added filter. The filter evaluates whether performing (or not performing) an action would violate the norms. If that is the case, the method then changes the utility value of the affected action to either 0 or to the highest possible +1 depending on the

violation and the norm. In the experiment, we are using three norms: (a) cars are not allowed to drive on the wrong side (lane in the wrong direction), (b) cars must drive on the left lane unless they are overtaking and (c) the maximum allowed speed is 55.

These were selected from the norms governing English roads. There was no particular norm analysis or selection process involved. It was, however, intuitively felt that these would minimize the number of crashed cars. The maximum speed norm was introduced after some preliminary experiments showed that this minimizes the number of crashed agents. In the second setup, the norms are: (a) if the light is not green do not enter junction, (b) from the left lane, the agent can choose to go left or straight ahead, (c) from the right lane, the agent can choose to go right and straight ahead. These norms ensure the smooth functioning of the system by removing any possibility of clashes in the junction.

For each time step
Evaluate all possible actions and assign them functionality values
Adjust the functionality values according to the norms
Discover the highest obtained functionality value for this time step
Perform the action with highest value and hierarchy

4.3 Rights Architecture

In setup one, rights-based agents use normative architecture as a base. Here, however, we are now using rights instead of norms. Looking at this as a right- hierarchy we have: (1) Right to live – do not do anything that could put you or others in danger, (2) Right to drive on your side – an agent on the correct side has the right not to be obstructed by agents going in the opposite direction, (3) Right to overtake – if the agent in front is slower than this agent then this agent has the right to overtake, (4) Right to use full speed – if this is not in conflict with previous rights, (5) Right to drive on the road – if this is not in conflict with previous rights. These rights (except the first one) correspond to the norms defined earlier although not perfectly as it is not possible to make a perfect translation. Right 1 is the most important one as it states that safety is paramount and thus allows or disallows invocation of any other rights. It will also force the slowing down or accelerating in dangerous situations (sometimes driving away from a dangerous situation is the safest way out). The agents use rights by evaluating what rights apply to them and to their neighbours in the current situation and then selecting an allowed action.

As stated earlier, in setup one, we wanted the systems to be based on the same basic architecture. We achieved this by extending the normative architecture with a method that evaluates the situation and from the agent rights' perspective (with safety as the main right) and either allows or disallows certain actions (depending on the rights in the situation). The second experimental setup uses two sets of rights that work independently of each other. The first set describes the rights with regards to entering the junction, the second one describes the rights with regard to the direction the car can take in junction. Even here the agent will evaluate the rights of itself and its neighbours and makes a decision on its next action depending on the evaluation. The first set of rights is: (1a) right to live, (2a) right to enter junction if the light is green and (3a) right to drive in the junction. The second set is (1b) if first right lane – the agent has the Right to go straight and right and, (2b) if first left lane – the agent has the Right to go straight and left.

For each time step
Evaluate all possible actions and assign them functionality values
Adjust the functionality values according to the norms
Evaluate each adjusted action and change the value according to rights
Discover the highest obtained functionality value for this time step
Perform the action with highest value and hierarchy

5 Experimental Parameters

All the experimental results are based on 100k time steps for each experiment, where the data for each 10 steps is averaged and saved for analysis, making 10k data points for each experiment. In total, we have 8 experimental scenarios for each experimental setup. Each experimental scenario is tested three times. Table 2 shows how the parameters (entry rate) change in each scenario. If the data for each experiment (in a single scenario) are consistent with the remaining two, the experimental results are then averaged into one set. In a situation where the results would not be consistent, more experiments would have been run. In the first experimental setup, the lanes run in both directions. In the second setup, the junction is connected to 4 roads, each with lanes in both directions. To compare the three methods, we decided to work with two different parameters:

- **The number of lanes in each direction**. This parameter was chosen since the number of lanes affects both behaviour and throughflow (efficiency) in a traffic system.
- **The rate of incoming agents**. The number of entering cars obviously affects traffic flow (complexity). The same architecture might perform differently depending on the complexity of the situation. Having more agents is testing the stability of the system.

Table 2. Experiment Layout for each Architecture

Entry Rate	500	300	100	50
Single Lane	3 exp	3 exp	3 exp	3 exp
Double Lane	3 exp	3 exp	3 exp	3 exp

The parameters were selected after considering real life traffic scenarios. What often changes is the number of lanes and the number of incoming cars. In order to analyse the results, we have decided to measure the following facts about the system: average speeds and number of crashed cars. Both of these figures describe the system and how it is performing in a specific situation. As in any scenario, we want to avoid the crashes as much as possible while maintaining as high speed as possible. Our main goal is to survive! In situations where survival rates are comparable, we will compare the agents' average speeds.

6 Results

The following are the results. Tables 3-6 give basic information about the system performance. Entry rate states than an agent will enter the system every 500, 300, 100

etc time steps depending on the experiment. The graphs show the average speeds for the systems and are used to show system behaviour graphically.

Table 3. Single Lane Experiments. AS=Average speed, CC-Crashed Cars, GL=Grid Lock

Entry Rate	500		300		100		50		10	
	AS	CC	AS	CC	AS	CC	AS	CC	AS	CC
RCT	53	112	53	364	GL	GL	GL	GL	GL	GL
Normative	44	0	44	0	43	0	41	0		
Rights	52	0	50	0	44	0	41	0		

Table 4. Double Lane Experiments. AS-Average speed, CC=Crashed Cars, GL=Grid Lock

Entry Rate	500		300		100		50		10	
	AS	CC	AS	CC	AS	CC	AS	CC	AS	CC
RCT	54	79	53	224	51	2335	49	7048	GL	GL
Normative	48	0	47	70	47	1560	45	4545		
Rights	51	0	49	28	47	1210	47	4707		

Table 5. Single lane entry to Junction, 4 roads connected to junction. AS=Average speed, CC-Crashed Cars, GL=Grid Lock

Entry Rate	500		300		100		50		10	
	AS	CC	AS	CC	AS	CC	AS	CC	AS	CC
RCT	29	55	21	67	6	296	GL	GL	GL	GL
Normative	12	0	5	0	0	0	0	0	0	0
Rights	39	0	11	0	0	0	0	0	0	0

Table 6. Double lane entry to Junction, 4 roads connected to junction. AS=Average speed, CC-Crashed Cars, GL=Grid Lock

Entry Rate	500		300		100		50		10	
	AS	CC	AS	CC	AS	CC	AS	CC	AS	CC
RCT	19	210	10	298	2	2366	0	8514	GL	GL
Normative	4	10	2	15	0	37	0	37	0	
Rights	7	6	2	4	0	30	0	45	0	

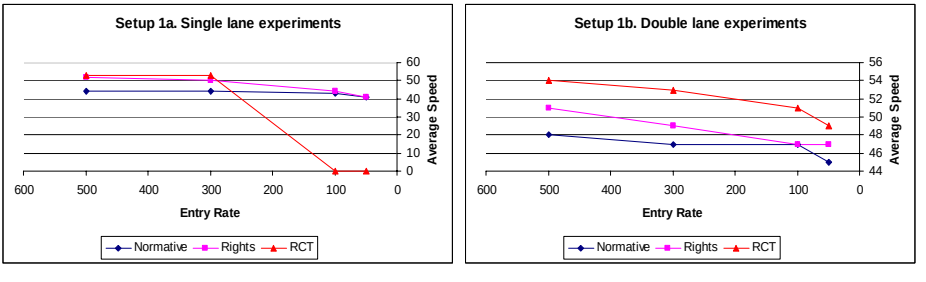

Fig. 1. Agent performance in experiment setup 1. In the experiments, agents are driving on a road with lanes in both directions. In the single lane scenario there is only 1 lane in each direction, in double there are 2.

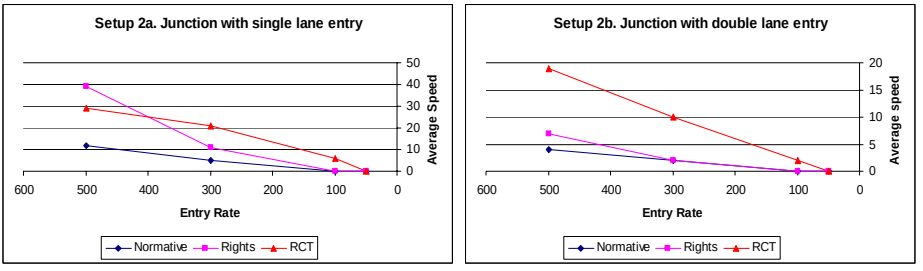

Fig. 2. Agent performance in experimental setup 2. In the experiments, a single lane entry means that for each of the four roads connecting to the junction there is only one lane that goes to the direction of the junction, in the double lane scenario there are two.

7 Analysis

The results show that both Normative and Rights based architectures outperform the Free-rider in each case. Even though the average speed of the Free-rider system is higher than the corresponding Rights and Normative systems, we can clearly see that when it comes to efficiency these systems still outperform the RCT since the number of crashed agents is a lot smaller.

In a single lane 1/500 experiment (Table 3), the number of entering cars is 400 (200 from each side). Using RCT, the number of crashed cars makes 20% of the total number of entered cars. One fifth of all the agents will not accomplish their goal meaning that, even though the average speed is higher than in the other two systems, the efficiency is 1/5 lower. At the same time, neither the Normative or the Rights system's have any crashed cars. The Rights systems average speed is almost 20% more than the Normative agents and very close to the RCT. In the next scenario, single lane 1/300, the number of entered cars is 667. In the RCT system over 50% of the agents crash. The Normative and Rights systems have again 0 crashes. Even in this case, the average speed of the Rights system is well above the speed of the Normative system. In the third single lane scenario, 1/100, we see a smaller difference between the Normative and Right based systems. The RCT agent based system cannot handle the number of agents and ends up in a grid lock situation.

The final single lane scenario shows no difference between Rights and Normative systems. In the Free rider scenario, we see again grid locks blocking the whole system.

In the double lane scenarios (Table 4) we again see similar results. In 1/500 experiments the RCT system is faster than both Normative and Rights system. However, approximately 10% of all Free rider agents crash whereas both other architectures have no crashes.

The next double lane scenario, 1/300, sees the rise of dead agents to 16% for the RCT system. At the same time, we also notice agents crashing in the Normative and the Rights systems. 5% Normative agents never reach their destination while only 2% of Right based agents crash. This is still outperforming the free-rider agents as we are interested in stability and efficiency and having 16% crashes is a lot more than 5% or 2%. In the third double lane scenario, 1/100, we continue to see similar trends to the previous two situations. The RCT system has now over 60% failure rate while the

Normative system has a failure rate of 39% and the Rights system 30%. RCT fails twice as often as the Rights system! Speed wise Norm based and Right based systems are doing the same (Fig. 2). When it comes to failure rate, Right system exceeds Norms by 25%.

In the final scenario, 1/50, the vast majority (88%, calculated from the total number of cars that enter) of RCT agents crash while the same figures for Normative architecture is 57% and 59% for the Rights system. This rather unexpected result will have to be looked into in more detail.

As we are looking for efficient and stable results, any experiments with a large number of crashed agents will automatically be assumed to be underperforming. When we compare the three architectures, we can clearly see that in most cases the Rights system is the best with regards to efficiency and stability. The free agent system is faster only because many agents crash. However, since we are interested is the survival of as many agents as possible, the average speed of the system becomes less interesting and is only used for comparison when agent failure rates are the same.

In single lane scenarios at levels of 1/100 the RCT agents end up in situations that cannot be resolved and the whole system locks with throughput 0. This obviously leads to the conclusion that the stability and efficiency is a lot more difficult to obtain in RCT systems. At the same time, both the normative and rights system continue functioning. As the failure rates are the same (0) we then compare the average speeds.

In the beginning the difference between the two is quite large (up to 20%, average speeds 44 and 52, Fig. 1 and table 3) in favour of the Rights system. As the number of cars entered per time unit increases, the rights agent results (the average speed) are converging toward the results of normative system. This is expected and explained by the fact that when the environment becomes more hostile (more cars using the road simultaneously) the right to "not being obstructed by other agents" is used a lot more.

The fact remains however that the Rights system is more, and in the worst case scenario just as, efficient as Normative system. In double lane scenarios, we see this even more clearly since the failure rate for the Rights system is significantly lower than in the Normative one. In dynamic MA Systems, we want autonomous agents to obtain the best stable results using as few resources as possible. Any agent that fails is a waste of resources. We are therefore interested in as high a survival rate as possible. The results clearly state that in most scenarios the Rights system will be the most successful one.

8 Discussion and Insights

So what do these results mean? We have already established earlier that a resulting comparison should not be taken as it is but rather a behaviour pattern needs to be discovered. If we look at the graphs and the behaviour of the systems, we can clearly see some patterns emerging. In the RCT system, the more complex the scenario becomes the worse the system performs. In non-complex scenarios the RCT system will on average perform better that the normative one, the reason being that RCT does not have to follow any behaviour constraining rules. In a situation with only one agent there is no risk for crashes and the agent does not have to take into account anything else. It can therefore use its full potential. In a complex situation, however, the free

choice means that agents cannot have full knowledge of how others will behave. This results in crashes. For non-complex systems, RCT will perform extremely well. On the other hand, in a normative system, we see little difference between very complex and non-complex scenarios. The system performance worsens only marginally when the complexity becomes higher. In a non-complex scenario, the agents will not perform at the peak of their capabilities and the system efficiency will not be utilized to its maximum. The norms ensure that the agents always perform the same. The rights-based system behaves differently to the other two. In non-complex situations, it behaves like a RCT system and in very complex situations it behaves like a normative one. As complexity increases, the behaviour of a rights system converges towards a normative one. This can be illustrated with a single car driving on a road. When there is no one else that could be affected by a car's actions, the car will drive as fast as possible. In a more complex system, the rights of others might outweigh the rights of this agent. It will therefore adjust its behaviour to others just as agents do in normative systems with the difference that for each time step the particular norm set might be different. In the rights system, the rights are flexible. Different rights will be applied depending on the complexity of the situation. The more complex a situation is, the higher hierarchy rights will be used. This means that the system as a whole changes its behaviour depending on what is best for it. A rights-based system can behave like a RCT or a normative system depending on the circumstances and what is most effective. If we generalize, a RCT agent needs to have a complete knowledge of all other agents and what they plan to do, in order to decide what to do next. A rights-based agent only needs to evaluate what rights the agents in its immediate neighbourhood have. A normative agent does not need to evaluate anything as its actions and everyone else's are predetermined. This means that the design complexity of a Rights system is somewhat higher than originally expected in [5], [6] and [11] and is in line with [12] (Table 7). It would be interesting to investigate further design and decision making complexity and map out how complexity of the environment affects the decision making complexity for different architectures.

Table 7. Reviewed comparison of RCT, Norms and Rights

	RCT	Norms	Rights
Complexity	High	Low	Low/Medium
Efficiency	Low	High	High
Stability	High	High	High
Flexibility	High	Low	High

9 Conclusions and Further Work

We have presented an empirical comparison of free, normative and rights-based agent coordination mechanisms in a simple car traffic simulation scenario. Rights give a system flexibility to perform more efficiently. In non-complex situation it allows the agent to behave like RCT and in very complex scenarios the agent will behave like a normative one. This flexibility between the two extremes and a range of in between

stages and the fact that the system adjusts itself make Rights a very promising alternative to RCT and norms. Further work will focus on more complex scenarios with better defined behaviours, more norms and more rights as well as other types of social norms.

References

1. Reiner R.: Arguments against the possibility of perfect rationality. Minds and Machines, 5, (1995) 373-389
2. Shoham, Y., Tennenholtz, M.: On the synthesis of useful social laws for artificial agents societies. In: Proceedings of the Tenth National Conference on Artificial Intelligence, AAAI-92, AAAI Press. Menlo Park, CA (1992) 276-281
3. Walker, A., Wooldridge, M.: Understanding the emergence of conventions in multi-agent systems. In: Proceedings of the First International Conference on Multi-Agent Systems, ICMAS-95. MIT Press. Cambridge, MA (1995) 384-389
4. Excelente-Toledo, C.B., Jennings, N.R.: The dynamic Selection of Coordination Mechanisms in Autonomous Agents. In: Multi Agent Systems, Vol. 9 (2004) 55-85
5. Alonso E.: Rights and Argumentation in Open Multi-Agent Systems. In: Artificial Intelligence Review 21(1) (2004) 3-24
6. Alonso E.: A Formal Theory of Rights and Argumentation. In: Open Normative Multi-Agent Systems. In W. Zhang and V. Sorge, (Eds.), Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems, Frontiers in Artificial Intelligence and Applications 112, IOS Press (2004) 153-167
7. Tennenholtz, M.: On stable social laws and qualitative equilibria. Artificial Intelligence. 102 (1998) 1-20
8. O'Hare G.M.P., Jennings, N.R. (Eds.): Foundations of Distributed Artificial Intelligence. John Wiley and Sons. New York, (1996)
9. Sycara K.P.: Multiagent Systems. AI Magazine, Vol. 19 (1998) 79-92
10. Weiss G.: Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence. MIT press, Cambridge, MA (1999)
11. Kristoffersson, P., Alonso, E.: Experimental Comparison of Rational Choice Theory, Norm and Rights based Multi Agent Systems. In: Proceedings of the Agent-Oriented Information Systems, AOIS-2005 (2005)
12. Kristoffersson, P., Alonso, E.: Rights for coordination in MAS: an experimental approach. In: proceeding of the 11th Conference of the Spanish Association for Artificial Intelligence, CAEPIA'11 (2005)

Adapted Information Retrieval in Web Information Systems Using PUMAS

Angela Carrillo-Ramos, Jérôme Gensel, Marlène Villanova-Oliver, and Hervé Martin

Laboratory LSR-IMAG. B.P. 72
38402 Saint Martin D'Hères Cedex, France
{carrillo, gensel, villanov, martin}@imag.fr

Abstract. In this paper, we describe how *PUMAS*, a framework based on Ubiquitous Agents for accessing *Web Information Systems* (*WIS*) through *Mobile Devices* (*MD*), can help to provide nomadic users with adapted information. Using *PUMAS*, the information delivered to a nomadic user is adapted according to, on the one hand, her/his preferences, intentions and history in the system and, on the other hand, the limited capacities of her/his *MD*. The adaptation performed by *PUMAS* relies on *pieces of knowledge* (we call "*facts*"), which are stored in *Knowledge Bases* managed by *PUMAS* agents. We focus here on the facts exploited to achieve adaptation by two of the four *Multi-Agent Systems* (*MAS*) that constitute the architecture of *PUMAS* (the *Information* and the *Adaptation MAS*). We also present an example which illustrates how *PUMAS* works and considers these facts when processing a query.

Keywords: PUMAS, Adaptation, Web Information System, Mobile Devices, Information Retrieval, Agent, Knowledge, Fact.

1 Introduction

Web-based Information Systems (*WIS*) are systems that permit collection, structuring, storage, management and diffusion of information, like traditional *Information Systems* (*IS*) do, but over a Web infrastructure. A *WIS* provides users with complex functionalities that are activated through a Web browser in a hypermedium interface. Nowadays, *Mobile Devices* (*MD*) can be used as devices for accessing a distant *WIS* but also as storage devices for (simple) *WIS* or applications. Thus, a *WIS* which executes on *MD* allows access, search and storage of resources (files) located on these *MD*.

However, having to cope with the limited capacities of *MD* (e.g. size of screen, memory, hard disk), *WIS* designers must use mechanisms and architectures in order to efficiently store, retrieve and deliver data using these devices. The underlying challenge is to provide *WIS* users with useful information based on an *intelligent search* and a *suitable display* of delivered information. In order to reach this goal, a *Multi-Agent System* (*MAS*) constitutes an interesting approach. The *W3C* [1] defines an *agent* as "*a concrete piece of software or hardware that sends and receives messages*". These messages can be used to access a *WIS* and to exchange information. A *MAS* can be a useful tool for modelling a *WIS* due to the inherent properties of

M. Kolp et al. (Eds.): AOIS 2005, LNAI 3529, pp. 243 – 258, 2006.

agents like the defined, owned and acquired knowledge they manage, their ability to communicate with users or other agents, etc. Carabelea *et al.* [2] have defined a *MAS* as "*a federation of software agents interacting in a shared environment that cooperate and coordinate their actions given their own goals and plans*". Moreover, agents can be executed on the *MD* and/or migrate through the net, searching for information on different servers (or *MD*) in order to satisfy user's queries. This is the underlying idea of the *Mobile Agent* concept [3].

Rahwan *et al.* [4] recommend the use of agent technology in *MD* applications because agents that execute on the user's *MD* can inform the systems accessed by the user about her/his contextual information. However, in the case of a mobile user, the agent must consider the fact that the changing location could produce changes in user tasks and information needs. Then, the agent also has to be proactive, and has to reason about user goals and the way they can be achieved.

Applications running on the *MD* (and their agents) must allow users to consult data at any time from any place. This is the underlying idea of *Ubiquitous Computing* (*UC*) [1]. Shizuka *et al.* [5] have stressed the fact that *Peer to Peer* (*P2P*) computing is one of the potential communicative architectures and technologies for supporting *ubiquitous/pervasive* computing. We can consider a *MAS* as a *P2P System*, since an agent is an inherent peer, because it can perform its tasks independently from the server and other agents. *P2P* systems [6] are characterized by i) a direct communication between peers with no communication needed through a specific server, and ii) the autonomy a peer gets for accomplishing some assigned tasks.

Concerning adaptation, special attention is paid to user's location in her/his profile. In order to provide the nomadic user only with *relevant information* (i.e. "*the right information in the right place at the right time*"), Thilliez *et al.* [7] have proposed "*location dependent*" *queries*, which are evaluated according to the user's current physical location (e.g. "*which are the restaurants located in the street where the user is?*"). Our work focusses also on this kind of *queries.*

Regarding adaptation to the reduced capacities of the *MD*, one objective is to anticipate the fact that some retrieved information cannot eventually be properly displayed (e.g. *MD* may not support a cumbersome format file). It is necessary to anticipate such situations at design time in order to decide which solution to implement. For instance, considering a query whose result contains video data, the corresponding result may not be delivered if the user accesses the *WIS* through a mobile phone that cannot display videos. In that case, the Negotiation vocabulary proposed by Lemlouma [8] can be used for adaptation purposes. It permits description of the user's *MD*, considering constraints in terms of network, software and hardware.

Many technical and functional aspects have to be considered when designing a *WIS* accessed through *MD*, especially when addressing the issue of adaptation of delivered information to the nomadic user [6, 7]. The goal of our work is to provide nomadic users who access a *WIS* through a *MD* with the most relevant information according to their preferences, but also according to their contextual characteristics and to the features of their *MD*. In [9], we have defined *PUMAS*, a framework for retrieving information distributed among several *WIS* and/or accessed through different types of *MD*. The architecture of *PUMAS* is composed of four *MASs* (a *Connection MAS*, a *Communication MAS*, an *Information MAS* and an *Adaptation MAS*), each one encompassing several *ubiquitous agents* which cooperate in order to achieve the

different tasks handled by *PUMAS* (*MD* connection/disconnection, information storage and retrieval etc.). In *PUMAS*, data representation, agent roles and data exchange are ultimately based on *XML* files (using *OWL*[1]). Through *PUMAS*, our final objective is to propose and build a framework which is, beyond the management of accesses to *WIS* through *MD*, also in charge of performing some adaptation processing over information. Users equipped with *MD* can use the *PUMAS* central platform in order to communicate together by means of agents that execute on their *MD*, or in order to exchange information (user's contextual information). In our case, users communicate through an *Hybrid P2P system*.

This paper is structured as follows. We first describe in Section 2 the architecture of *PUMAS*. The main contributions of this paper are, on the one hand, the definition of *pieces of knowledge* (that we call *facts*) used for adaptation purposes by *PUMAS* agents, especially those belonging to the *Information* and to the *Adaptation MAS*, and, on the other hand, the data representation based on *XML* files. In Section 3, we present a scenario that shows how *PUMAS* processes a query submitted to the system. An example that illustrates our proposition is given in Section 4. We discuss works related to *PUMAS* in Section 5 before we conclude in Section 6.

2 The PUMAS Framework

In this section, we present the architecture of *PUMAS*, its four *MASs*, their relations and, the data exchange and communications they perform in order to achieve adaptation of information for the user.

2.1 An Overview of the PUMAS Architecture

The architecture of *PUMAS* is composed of four *MASs* (see Fig. 1 for the logical structure of *PUMAS*). Firstly, the *Connection MAS* provides mechanisms for facilitating connection from different types of *MD* to the system. Secondly, the *Communication MAS* ensures a transparent communication between *MD* and the system, and applies a *Display Filter* to display the information in an adapted way according to technical constraints of the user's *MD*. To achieve this, it is helped by agents of the *Adaptation MAS*. Thirdly, the *Information MAS* receives users' queries, redirects them to the "*right*" *WIS* (e.g. the nearest *WIS*, the more consulted one), applies a *Content Filter* (with the help of the *Adaptation MAS* agents) according to the user's profile in the system and returns results to the *Communication MAS*. Finally, the *Adaptation MAS* communicates with agents of the three other *MAS* in order to provide them with information about the user, connection and communication features, *MD* characteristics etc. The services and tasks of its agents essentially consist of managing specific *XML* files that contain information about the user and the device. These agents also have some knowledge, which allows them to select and to filter information for users. This knowledge comes from analysis of the user's history in the system (e.g. last connections, queries, preferences).

---

[1] *OWL*: *Ontology* Web *Language* builds on *RDF* and *RDF Schema* and adds more vocabulary for describing properties and classes (relations between classes, cardinality, equality, richer typing of properties, characteristics of properties and enumerated classes). *http://www.w3.org/2004/OWL/*

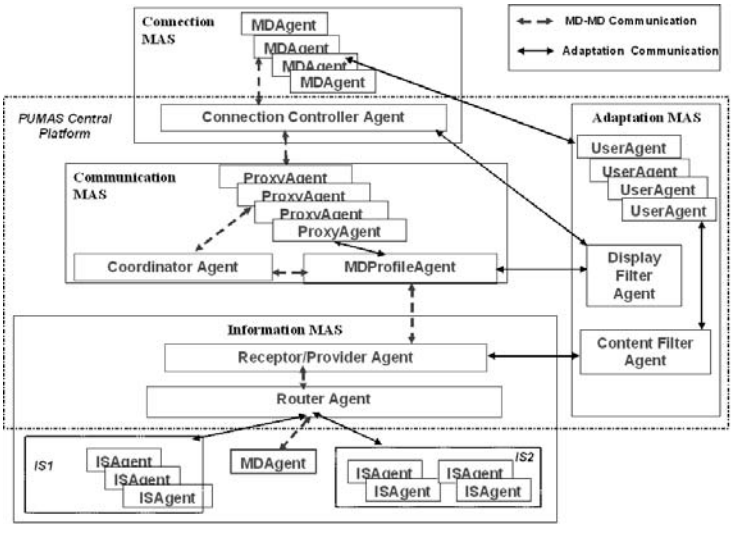

Fig. 1. The *PUMAS* Architecture

The inherent mobility of nomadic users is supported by *ubiquitous agents*: the *Mobile Device Agents* executed on the user's *MD* and the *ISAgents* executed on the same device than the *WIS* to which they belong. Such ubiquitous agents retrieve some needed information and can communicate with other agents to perform tasks. The *Hybrid P2P Architecture* of *PUMAS* copes with the following issues: security in applications (security problems inherent to agent mobility), communication between agents in a point to point or in a broadcast way, management of the status of the agent (e.g. connected, disconnected and killed) and its services. In the following subsections, we describe the tasks achieved by each *MAS* of *PUMAS*.

2.2 The Connection MAS

This *MAS* includes several *Mobile Device Agents* (*MDA*) and one *Connection Controller Agent* (*CCA*).

The ***Mobile Device Agent*** is executed on the user's *MD*. Its knowledge is composed of general rules of behaviour and characteristics related to the type of *MD* used (e.g. *PDA*) as well as some specific roles defined according to the application (e.g. this agent is used for transmitting a file). The *Mobile Device Agent* manages a *XML* file (*Device Profile XML* file, located on the user's *MD*), which describes *MD* features and shares this information with the *Display Filter Agent* (belonging to the *Adaptation MAS*) through the *Connection Controller Agent* (the *Mobile Device Agent* sends this file to the *Connection Controller Agent* – executing on the central platform of *PUMA* – and the latter exchanges this information with the *Display Filter Agent*). This file contains some information about the requirements of the application, network status, hypermedia files supported by the *MD*, conditions for disconnecting: inactive session for more than *X* minutes, disconnection type (e.g. willingly, automatic), etc. One *Mobile Device Agent* also manages another *XML* file, which describes characteristics of the user's session (using *OWL*, see Fig. 2): who is the user

connected (user *ID*), when the session began and what is the *MD* connected (beginning time, *CurrentMD*). This file will be sent to the *UserAgent* (belonging to the *Adaptation MAS*):

```
<?xml version="1.0"?>
  <rdf:RDF…  …
      <owl:Ontology rdf:about=""/>
          <owl:Class rdf:ID="SessionProfile"/>
          <owl:Class rdf:ID="CurrentUser">
              <rdfs:subClassOf rdf:resource="#SessionProfile"/></owl:Class>
          <owl:Class rdf:ID="BeginningTime">
              <rdfs:subClassOf rdf:resource="#SessionProfile"/></owl:Class>
          <owl:Class rdf:ID="CurrentDevice">
              <rdfs:subClassOf rdf:resource="#SessionProfile"/></owl:Class>
  </rdf:RDF>
```

Fig. 2. Code excerpt of the *User's Session XML* file

The ***Connection Controller Agent*** executes on the central platform of *PUMAS* and gets the user's location and *MD* type (e.g. *PDA*) from the *User Location XML file* (which contains the physical and logical user's location features; this file is also defined using *OWL*) and from the *Device Profile XML file* (which describes the features of the *MD*), respectively. Both files are provided by the *Mobile Device Agent* and locally managed by the *Connection Controller Agent.* The latter serves as an intermediary between the *Connection MAS* and the *Communication MAS.* It also checks connections established by users and the status of agents (e.g. connected, disconnected, killed), and links each *Mobile Device Agent* to its corresponding *Proxy Agent* in the *Communication MAS* (see next section).

The *XML files* (*User Location*, *Session* and *Device Profile XML* files) managed by the *Mobile Device Agent* and the *Connection Controller Agent* have been defined using extensions introduced by Indulska *et al.* [10] to *CC/PP* [1]. These extensions include some user's characteristics like her/his location, application requirements, session features (e.g. user, device, application) and the profile of the *MD* in order to provide a complete description of the user and her/his *MD*.

2.3 The Communication MAS

This *MAS* has an interface that makes communication between users transparent and activates the mechanism for displaying the information according to the features of the *MD*. It is composed by several *Proxy Agents* (*PA*), one *MDProfile Agent* (*MDPA*) and one *Coordinator Agent* (*CA*). These agents execute on the central platform of *PUMAS.*

There is one ***Proxy Agent*** for the connection of each *Mobile Device Agent.* Two different users can connect themselves to the system through the same *MD*, which leads to two different *Proxy Agents* and two different sessions. The main task of a *Proxy Agent* is to represent a *Mobile Device Agent* within the system. In this case, there are two agents: one *Mobile Device Agent* in the *MD* and one *Proxy Agent* in the central platform of *PUMAS*.

The ***MDProfile Agent*** has to check the user's profile (according to her/his *MD*) and her/his information needs. In addition, this agent together with the *Coordinator Agent* defines and checks the mechanism that sends, for example, hypermedia data to the user. If the user's request has, as a result, several images, these agents define the order and number of images to be shown by the screen according to the capabilities of the user's *MD*. The *MDProfile Agent* also shares information about specific *MD* features for the user's session with the *Display Filter Agent* (belonging to the *Adaptation MAS*).

The ***Coordinator Agent*** is in permanent communication with the *Connection Controller Agent* in order to verify the connection status of the agent that searches for information. The *Coordinator Agent* knows all the agents connected in the system thanks to *XML* files managed by the *Mobile Device Agent* (through its *Proxy Agent*). If there are some problems with the *Connection Controller Agent* (e.g. if the *Connection Controller Agent* fails or if there is a lot of connections), the *Coordinator Agent* can play the role of the *Connection Controller Agent* until the problems are fixed. At that moment, the *Connection Controller Agent* and the *Coordinator Agent* must synchronize the information about connected agents and check current connections.

A more detailed description of the *Connection* and the *Communication MAS* can be found in [9]. The main contribution of this paper, described in the next section, deals with the description of the knowledge managed by the *Information* and the *Adaptation MAS* agents in order to support the adaptation capabilities of *PUMAS*.

2.4 The Information MAS

The *Information MAS* is composed of one or several *Receptor/Provider Agents* (*R/PA*), one or several *Router Agents* (*RA*) and one or several *ISAgents* (*ISA*).

A ***Receptor/Provider Agent*** that is located in the central platform of *PUMAS* owns a general view of the whole system. It knows agents of both the *Communication* and the *Information MAS*. The *Receptor/Provider Agent* receives all requests that are transmitted from the *Communication MAS* and redirects them to the *Router Agent*, which is in charge of finding the "*right*" *WIS* in order to execute the query. Once a query has been processed by the *ISAgents*, the *Receptor/Provider Agent* checks whether query results consider the user's profile (i.e. preferences, user's history) by means of the *Content Filter Agent* (belonging to the *Adaptation MAS*).

In order to redirect a query to the "*right*" *WIS*, a ***Router Agent*** (which executes on the central platform of *PUMAS*) applies a strategy that depends on one or several criteria: user's location, peer similarity, time constraints, user's preferences etc. The strategy can lead to sending the query to a specific *WIS*, to sending the query using broadcast and/or to the division of the query in sub-queries, each being sent to one or several *WIS*. A *Router Agent* is also in charge of compiling results returned by the *WIS* and of analyzing them (according to the defined criteria) to decide whether the whole set of results or only a part has to be sent to a *Receptor/Provider Agent*.

The *Router Agent* stores in its *Knowledge Base pieces of knowledge* (that we call *facts* and describe below using *JESS*[2]) for each *WIS*. One *fact* is made up of the

[2] *JESS* is a rule engine and scripting environment that enables building Java applications that have the capacity of "reasoning" using knowledge supplied in the form of declarative rules. *http://herzberg.ca.sandia.gov/jess/*

characteristics of the *WIS*, like its name, its managed information, the type of device on which it is executed (e.g. server, *MD*) and the agent (*ISAgent*) associated with this *WIS* and which can be asked for information. When the *Router Agent* has to redirect a user's query, it exploits these facts in order to select the *WIS*, especially, the *ISAgents* (which execute on the same device that the *WIS*) to which sub-queries have to be redirected. The following fact defines a *WIS* and is represented by a *JESS* template[3]:

(*deftemplate WIS* (*slot name*)
(*slot agentID*) (*slot device*)
(*multislot information_items*))

For instance, the following fact defines the *Pharmacy WIS* of a hospital. The *WIS* is called *PharmacyWIS* and it executes on a *server*. *PharmacyISA* is the *ISAgent* which executes on this *WIS*. The *PharmacyWIS* contains information about *medicines* and *patient prescriptions*:

(*assert* (*WIS* (*name PharmacyWIS*)
(*agentID PharmacyISA*) (*device server*)
(*information_items "medicines" "patient's_prescription"*)))

The location of the *WIS* could change, especially if this *WIS* runs on a *MD*. The *Router Agent* can be informed about the changes in the location of the *WIS* by means of the *ISAgents* that execute on these *WIS*.

In order to send (sub-) queries and analyse their results, the *Router Agent* must check the user's preferences (information provided by the *Content Filter Agent* via the *Receptor/Provider Agent*). The user's preferences are represented as facts defined as follows:

(*deftemplate User_Preference* (*slot userID*)
(*slot required_info*)
(*multislot complementary_info*)
(*multislot actionD*) *; actions for doing*
(*slot problem*)
(*multislot actionR*)) *; actions for recovering*

An ***ISAgent*** associated with a *WIS* (and which executes on the same device that the *WIS*) receives the user's query from the *Router Agent* and is in charge of searching for information. Once a result for the query is obtained, the *ISAgent* returns it to the *Router Agent*. An *ISAgent* can execute a query by itself or delegate this task to the adequate *WIS* component. This depends notably on the nature of the *WIS*. Our approach addresses complex and possibly a distributed *WIS* located on server(s) but also a very simple *WIS* that only relies on some files located on a *MD*. In this last case, one *ISAgent* may be sufficient to ensure the right functioning of the *Information MAS*. It is worth noting that, in this case, what we call an *"ISAgent"* is in fact the *Mobile Device Agent* of a *MD* that can play the role of an *ISAgent* since it has the knowledge required for executing a query on files stored in the *MD*. In a complex *WIS*, the *ISAgent* can collaborate with other *ISAgents* (if the *WIS* has been developed

[3] We define our pieces of knowledge using the syntax of the *JESS* unordered facts. We declare each unordered fact by means of the primitive "*deftemplate*". To define an instance of an unordered fact in *JESS* and store it into the *JESS Knowledge Base*, we use the primitive "*assert*".

following the *MAS* paradigm) or with any other *WIS* component to perform a query. In the case of a non *MAS* based *WIS*, our approach only requires that an *ISAgent* is developed in order to ensure the communication between *PUMAS* and the *WIS*.

2.5 The PUMAS Adaptation MAS

The adaptation capabilities of *PUMAS* rely on a two step filter process that aims at providing a user with adapted information according to both the user and her/his *MD*. First, the *Content Filter* allows selection of the most relevant information according to the user's profile defined. Second, the *Display Filter* is applied on the results of the first filter and considers characteristics and technical constraints of the user's *MD*.

The *Adaptation MAS* is composed of several *UserAgents* (*UA*), one *Display Filter Agent* (*DFA*) and one *Content Filter Agent* (*CFA*). These agents execute on the central platform of *PUMAS*.

```
<rdf:RDF …
<owl:Ontology rdf:about=""/>
        <owl:Class rdf:ID="UserProfile"/> <owl:Class rdf:ID="Beliefs">
                <rdfs:subClassOf rdf:resource="#UserProfile"/></owl:Class>
        <owl:Class rdf:ID="Intentions">
                <rdfs:subClassOf rdf:resource="#UserProfile"/></owl:Class>
         <owl:Class rdf:ID="User">
                <rdfs:subClassOf rdf:resource="#UserProfile"/></owl:Class>
        <owl:Class rdf:ID="Preferences">
                <rdfs:subClassOf rdf:resource="#UserProfile"/></owl:Class>
</rdf:RDF>
```

Fig. 3. Code excerpt of the *User Profile XML* file

Each ***UserAgent*** manages a *XML* file (*User Profile XML* file, see Fig. 3) that contains personal characteristics of the user (e.g. user ID, location) and her/his preferences (e.g. the user wants only video files). This file is obtained by means of the *Mobile Device Agent* (this file is managed by the *UserAgent* and updated by the *Mobile Device Agent*). There is only one *UserAgent* that represents a user at the same time (even though the user has two sessions at the same time through the same or different *MD*). Since a user can access the system through several *MDs*, the *UserAgent* communicates with the *Mobile Device Agents* and the *Proxy Agents* (which respectively belong to the *Connection* and the *Communication MAS*) to analyse and centralize all the characteristics of the same user. The *UserAgent* communicates with the *Content Filter Agent* to send the *User Profile XML* file. When the *Content Filter Agent* receives this file, it stores this information as facts in its *Knowledge Base* (this agent manages a registry of user's preferences). When the *Receptor/Provider Agent* (belonging to the *Information MAS*) asks the *Content Filter Agent* for the user's preferences, the latter sends it the latest *XML* file received from the *UserAgent*. If the *UserAgent* does not send this file (e.g. there are no user preferences for the current session), the *Content Filter Agent* considers the preferences from previous sessions.

We can establish that queries depend on one or several criteria for adaptation purposes: the user's location, her/his history in the system, activities developed during a time period, movement orientation, privacy preferences, etc. An *Adaptation_ Criterion* could be defined as:

(*deftemplate Adaptation_Criterion*
(*slot userID*) (*multislot criteria*) (*multislot attributes*))

An example of *Adaptation_Criterion* that expresses that all of *Doctor Smith's* queries depend on his location, especially when he is at the *North Hospital* could be:

(*assert* (*Adaptation_Criterion*
(*userID "Doctor Smith"*) (*criteria location*) (*attributes "North_Hospital"*)))

The ***Display Filter Agent*** manages a *Knowledge Base* that contains general information about features of different types of *MD* (e.g. format files supported) and acquired knowledge from previous connections (e.g. problems and capabilities of networks according to data transmissions). Each *MDFeature* is defined as a fact and represented as follows:

(*deftemplate MDFeature* (*slot MDtype*) (*multislot feature*))

where each feature is represented as a fact as follows:

(*deftemplate feature* (*slot type*) (*multislot causes*))

An example of a fact for a *MDFeature* which corresponds to file formats that are supported by a *Pocket PC hp IPAQ h5550* in different network types is shown as follows. We assume that it cannot support video sent on a *Wi-Fi Network* but it does support several images using either *Bluetooth* or a *Classical Network*:

(*deffacts MDFeature* (*MDType "PocketPC hpIPAQ h5550"*)
(*feature* (*type "video_not_supported"*) (*causes "Wi-Fi Network"*))
(*feature* (*type "several_images"*) (*causes "Bluetooth" "Classical Network"*)))

The ***Content Filter Agent*** manages a *Knowledge Base* that contains preferences, intentions and characteristics of users. The *User_Preference* fact is composed of a *userID* (which identifies the owner of this preference), required information (*required_info*) and complementary information (*complementary_info*). The last is added to the *User_Preference* definition by the *Content Filter Agent*, which analyses queries of previous sessions (e.g. information frequently asked). This fact is also composed of information describing what and how user would like answers from the system (to be presented to her/him) and in case there are any problems, what and how the system must answer (*list of actions for recovering*). In order to do this, each *action* is defined as a fact and represented as follows:

(*deftemplate action* (*slot name*) (*multislot attribute*))

In this definition, *name* refers to an action chosen between a defined list (e.g. "*show*", "*save*", "*transfer file*", "*cancel*") and each action has a list of *attributes*. For instance, the fact which represents the action "*show*" has for its properties the *order*, *format* and *type of the file*, is:

(*assert* (*action* (*name show*) (*attributes "order" " format" "file_type"*)))

Since an *attribute* can be complex, we define it as a fact:

(*deftemplate attribute* (*slot name*) (*multislot list*))

An example of *attribute* that defines the order in which information is displayed, could be:

(*assert* (*attribute* (*name order*)
(*list "patient's_tests" "patient's_diet" "patient's_prescribed_medicines"*)))

We define a *problem* as an event that is not desirable during the execution of an action or that is the cause of a failure (e.g. the *MD* cannot *show* an image). Each problem is defined as a fact and represented as follows:

(*deftemplate problem* (*slot name*) (*slot type*) (*multislot causes*))

where *name* corresponds to a description of the problem, the *type* can be chosen from a defined list (e.g. *incompatibility, unable IS, unable agent*) and the *causes* correspond to a list of causes of this problem (e.g. *MD* cannot support a specific format file, network problems). A fact, which defines the *problem* related to a specific user's location that is out of range of a wireless network and disables her/him from accessing the Internet, is:

(*assert* (*problem* (*name "out_range_connection"*) (*type "lack_of_access"*)
(*causes "user_located_out_of_range" " network_out_of_service"*)))

3 PUMAS Scenario

In this section, we present a scenario in order to show the interactions that take place between *PUMAS* agents when a query is submitted to the system.

When a user sends an information query *Q* (see Fig. 4), the *Mobile Device Agent* sends it to the *Connection Controller Agent.* Whenever this query is location and time dependent, the *Connection Controller Agent* introduces the *time of connection*, the *user's location* and the characteristics of the *user's MD* connection (these latter characteristics are exchanged with the *Display Filter Agent*) in query *Q* which leads to the production of a new query *Q'* (in Fig. 4, *Q'=Q + user's ST*) that is then sent to the *Proxy Agent*. The query passes by the *Coordinator Agent* and then by the *MDProfile Agent*. The latter adds to query *Q'* some features related to the *MD;* these features are provided by the *Display Filter Agent* which has previously learned them from previous queries or retrieved them from its *Knowledge Base*. The new query *Q"* (in the Fig. 4, *Q"= Q' + MD features*) is sent by the *MDProfile Agent* to the *Receptor/Provider Agent.* The *Receptor/Provider Agent* complements the query *Q"* with specific characteristics of the user in the system by requesting the *Content Filter Agent* (in Fig. 4, *Q'''=Q" + user's preferences, intentions, history*). The *Receptor/Provider Agent* sends the query *Q'''* to the *Router Agent,* which decides (according to the query, the system rules and the facts in its *Knowledge Base*) which are the *ISAgents* able to answer. It can send the query to a specific *ISAgent* or to several *ISAgents* (e.g. waiting for the first to answer) or, it can divide the query into sub-queries, which are sent to one or several *ISAgents.* The scenario in Fig. 4, shows for instance that query *Q'''* is divided into $Q^{'''-1.1}$, $Q^{'''-1.2}$, $Q^{'''-1.3}$ and $Q^{'''-1.4}$, which are sent to the *ISAgents* executed on a server and different *MD*.

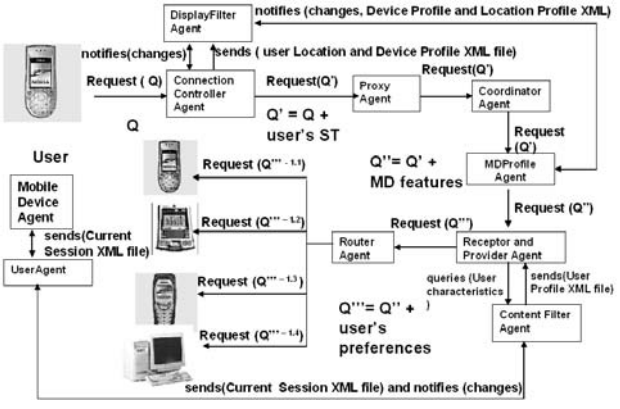

Fig. 4. Scenario of sending a query

When a user *U1* has an information query for another user *U2*, both equipped with *MD*, the query is propagated from the *Mobile Device Agent* executed on the *U1*'s *MD* towards the *Router Agent*, which redirects it to the *Mobile Device Agent* executed on the *U2*'s *MD*. This *U2*'s *Mobile Device Agent* changes its role to become an *ISAgent*, i.e. the agent in charge of answering information queries. This change of role is possible because a *Mobile Device Agent* has knowledge for managing information stored in the *MD* on which it executes and it has the capability of answering information queries.

4 Example

In this section, we illustrate processes performed by *PUMAS* agents using the example of a hospital *WIS*.

Let us suppose that doctors equipped with *MD* (e.g. *PDA*) access the information system of a hospital that is distributed between several *MD* and/or one or several *WIS* (see Fig. 5). Doctors can also receive information according to their location, preferences, technical characteristics of their *MD* and considerations about their connection time. For instance, when visiting a patient, doctors with *MD* can consult information about her/his clinical history, medical tests, prescriptions etc. By indicating the location of the patient (e.g. room, bed) and the current date (extracted from the system), the doctor can identify the patient and get her/his personal information. To do this, the application on her/his *MD* must consult different *WIS* of the hospital (e.g. pharmacy, doctors). Doctors could also communicate with other doctors (peers), through their *MD*, in order to get some advice or help (e.g. questions which can only be answered by the specialist doctor who has previously examined this patient).

When a doctor comes into the patient's room, she/he enters information about the location of the patient while the application gets the date of the system (information about the time). The *Mobile Device Agent* that executes on the doctor's *MD* sends the

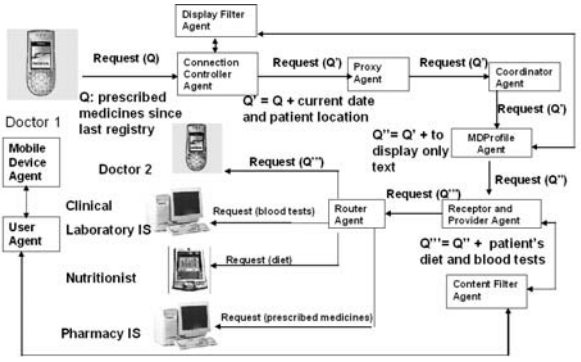

Fig. 5. Sending a query in the hospital *WIS*

query (*who is the patient?*). The query is propagated through the *PUMAS* core: it is first transmitted through the *Connection Controller Agent*, then to the *Communication MAS* agents (*Proxy Agent*, *Coordinator Agent* and *MDProfile Agent*). The *MDProfile Agent* can include, in the query, information according to the *MD* (e.g. this kind of *MD* cannot support graphical format but only text files. Then, if the doctor asks for the results of the test, she/he only could get them in a text format). For example, if the doctor has been connected through a *Palm Tungsten C*, the *MDProfile Agent* asks the *Display Filter Agent* for information about this *MD*. The *MDProfile Agent* could receive from *Display Filter Agent* facts defined as follows:

(*deffacts MDFeature* (*MDType "Palm Tungsten C"*)
(*feature* (*type "video_not_supported"*)(*causes "Wi-Fi_Network"*))
(*feature* (*type "several_images"*)(*causes "Wi-Fi_Network"*)
(*feature* (*type "text"*)(*causes "Wi-Fi_Network" "Classical_Network Bluetooth"*))) *;*

Then, the *MDProfile Agent* sends the query to the *Receptor/Provider Agent*, which can include in the query preferences previously expressed by the doctor. Those preferences are expressed in the *User Profile XML* file (see Section 2.5) and are translated as facts by the *UserAgent* and the *Content Filter Agent*. The following example corresponds to a doctor's preference: when a doctor says "*when asking for a blood tests, the system must also provide me with the patient's diet and the prescribed medicines. I do prefer graphical results but if my MD cannot support this format, I would like receive the results in text format*", this can be translated into the following fact of the *UserAgent*:

(*deffacts User_Preference*
(*userID "Doctor Smith"*)(*required_info "blood tests"*)
(*complementary_info "patient's_diet" "prescribed_medicines"*)
(*action show*) (*attribute* (*name order*)
(*list "patient's_tests" "patient's_diet" "patient's_prescribed_medicines"*))
(*attribute* (*name graphical_format*) (*list "JPEG"*))
(*problem* (*name "HyperMediaNotSupportedByMD"*)
(*type "incompatibility"*) (*causes "OnlyTextFileSupported"*)) (*attribute* (*name order*)
(*list "patient's_tests" "patient's_diet" "patient's_prescribed_medicines"*))
(*attribute* (*name text_format*) (*list "XML" "txt"*)))

The *UserAgent* transfers this information to the *Content Filter Agent*, which stores this fact and sends it to the *Receptor/Provider Agent.* The *Receptor/Provider Agent* adds this preference to the query and sends it to the *Router Agent*. The *Router Agent* receives the complete query and, with the information about the *WIS*, the *Router Agent* can split the query in sub-queries and redirect each one towards the appropriated *WIS*. The following facts are exploited in this example by the *Router Agent* in order redirect the queries to the *ISAgents* of the hospital's *WIS*:

(*assert* (*WIS* (*name LaboratoryWIS*)
(*agentID LaboratoryISA*) (*device server*)
(*information_items " test" " patient's_ test" "reactive"*)))

(*assert* (*WIS* (*name PatientDietWIS*)
(*agentID DietISA*) (*device MD*)
(*information_items "patient's_diet" "nutritionist's_appointments"*)))

The *Router Agent* redirects the query to the *ISAgent* located in the *WIS*, which manages information about patients in the hospital. All queries follow the same path from the *Mobile Device Agent* towards the *Router Agent*. If the doctor wants to know the *last medicines prescribed* to this patient, the *Router Agent* redirects the query to the *ISAgent* located in the *PharmacyWIS*. If the query concerns another *doctor* (*peer*), the *Router Agent* redirects the query to the *ISAgent* located in the peer's *MD*. A doctor can also ask for information about a specific patient to several of her/his peers. In this case, the *Router Agent* could send the query using broadcast or it could split the query according to the receiver peer (e.g. queries related to heart conditions for the cardiologist) or according to the defined criteria in the *User Profile XML* file (e.g. if the criterion of adaptation of the query is the *location*, queries must only be redirected to doctors at the *same* or *closed location* of the sender). Retrieved information is organized by the *Router Agent* (e.g. the last prescribed medicines, peer answers about this patient) and is returned to the doctor who has sent the query following the inverse path. The different agents have to check results because, for instance, the doctor may have been disconnected from the system (due to some network problems), and recovered her/his session in a new connection whose characteristics are different from the previous ones: it could be that she/he can now consult the system using another kind of *MD* that supports some graphical format (which constitutes a doctor's preference that can now be satisfied).

Through this example, we can observe the behaviour of the *Hybrid P2P Architecture* of *PUMAS*. The core of *PUMAS* centralizes queries: i) it is in charge of obtaining the most relevant information and, ii) it is in charge of applying the *Content* and *Display Filters* to adapt answers. The main peer characteristics of *PUMAS* agents are illustrated by the fact that, firstly, agents have the autonomy of connecting to and disconnecting from the system. Secondly, a *MD* can ask for a communication with a specific *WIS* (located on a server or on a *MD*) passing this information as a parameter of the query; the *Router Agent* transmits the query to this specific *WIS* which exemplifies an agent to agent communication (e.g. when doctors exchange information about a patient using their *MD*).

Another advantage offered by *PUMAS* is that it helps a user who does not know which specific *WIS* to ask for information to find the most appropriate one(s). The *Router Agent* redirects a query by means of an intelligent analysis of the query and the

help of the *ISAgents* that achieve an intelligent search inside the different *WIS* (pharmacy, laboratory, patients etc. in our example).

5 Related Works

In this section, we present some agent-based architectures or frameworks for adapting information to users.

Berhe *et al.* [11] proposes an architectural framework that exploits four profiles for adapting information content to a user: *content or media* (type, format, size, location where media is stored), *user* (preferences), *device* (hardware and software capabilities), *network* and *service* (supported media formats, network connection, bandwidth, latency performance). However, unlike *PUMAS*, this proposal does not consider information retrieval from different types of devices (servers and *MD*).

Sashima *et al.* [6] proposes an agent-based coordination framework for ubiquitous computing. It coordinates services and devices to assist a particular user in receiving a particular service in order to maximize her/his satisfaction. This framework assists users in accessing resources in ubiquitous environments. These authors consider contextual features of nomadic user, especially location. Unlike *PUMAS*, this framework does not consider the adaptation of information according to the access devices or the possible distribution of data among different devices.

The work of Gandon *et al.* [12] proposes a Semantic Web architecture for context-awareness and privacy. This architecture supports automated discovery and access of a user's personal resources subject to user-specified privacy preferences. Service invocation rules along with services ontologies and services profiles allow identification of the most relevant resources available to answer a query. However, it does not consider the information that can answer a query can be distributed between different sources.

CONSORTS Architecture [13] is based on ubiquitous agents and designed for a massive support of *MD*. It detects user's location and defines the user's profile to adapt the information to her/him. The *CONSORTS* architecture proposes a mechanism for defining relations that hold between agents (e.g. communication, hierarchy, role definition), with the purpose of satisfying user's requests. However, it does not consider the distribution of information between *MD* (which could improve response time) nor user's preferences.

PIA-System [14] is an agent-based personal information system for collecting, filtering and integrating information at a common point, offering access to information by *WWW*, e-mail, *SMS*, *MMS* and *J2ME* clients. It allows the user on the one hand, to search explicitly for specific information and, on the other hand, to be informed automatically about relevant information divided into slots (user specifies her/his working time and this divided the day in pre, work and recreation). A personal agent manages the individual information provisioning, tailored to user's needs according to her/his profile and current situation. However, a *PIA-System* only searches information in text format (e.g. documents). It does not take into account either the adaptation of different kinds of media to different *MD* or the user's location.

6 Conclusion

In this paper, we have presented *PUMAS*, a framework based on agents and the *P2P* approach. Peer characteristics of *PUMAS* appear in the cooperation developed by agents in order to store and retrieve information and in the possibility that two users, equipped with *MD*, communicate through the central platform offered by *PUMAS*. Its architecture relies on four *Multi-Agents Systems* (*MAS*) for the *Connection*, the *Communication*, the *Information* and the *Adaptation MAS*. *PUMAS* also benefits from the *P2P* characteristics of an *Hybrid P2P architecture*. *PUMAS* provides each agent with a mechanism for identifying, authenticating and recognizing its peers. This paper has focussed on the representation of the *pieces of knowledge* (called *facts*), stored in *Knowledge Bases* and used by *PUMAS* agents in order to perform their assigned tasks. We can highlight the *intelligent* and *adaptive information search* achieved by means of *PUMAS* agents. The search is *intelligent* because it is based on the knowledge of an agent and its capability of reasoning. It is also *adaptive* because it considers nomadic user's profiles, characteristics of her/his *MD* and features of the ubiquitous context.

Our future work concerns the implementation of each component (*MAS*) of *PUMAS*. We also need to define an extension of the current *ACL* that considers spatial-temporal (contextual) features and a strategy description language, as well as *Query Routing* mechanisms and algorithms [15] for the *Router Agent* in order to propagate queries towards the "*right*" *WIS* and to compile answers. Moreover, the mechanisms and strategies applied by the *PUMAS* agents (especially those belonging to the *Adaptation MAS*) in order to achieve the *Content* and the *Display Filters* also have to be precisely defined.

Acknowledgments. The author Angela Carrillo-Ramos is partially supported by Universidad de los Andes (Bogotá, Colombia).

References

1. *http://www.w3.org/TR/webont-req/* (L.A.: August 2006).
2. Carabelea, C., Boissier, O., Ramparany, F.: Benefits and Requirements of Using Multi-agent Systems on Smart Devices. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.): Proc. of the European Conference on Parallel Processing. Lecture Notes in Computer Science, Vol. 2790, Springer-Verlag, Berlin Heidelberg New York (2003) 1091-1098.
3. Lin, F.C., Liu, H.H.: MASPF: Searching the Shortest Communication Path with the Guarantee of the Message Delivery between Manager and Mobile Agent. In: Yang, L.T., Guo, M., Gao, G.R., Jha, N.K. (eds.): Proc. of the Conference on Embedded and Ubiquitous Computing. Lecture Notes in Computer Science, Vol. 3207, Springer-Verlag, Berlin Heidelberg New York (2004) 755-764.
4. Rahwan, T., Rahwan, T., Rahwan, I., Ashri, R.: Agent-Based Support for Mobile Users Using AgentSpeak(L). In: Giorgini, P., Henderson-Sellers, B., Winikoff, M. (eds.): Proc. of the 5th International Bi-Conference Workshop on Agent-Oriented Information Systems. Lecture Notes in Artificial Intelligence, Vol. 3030, Springer-Verlag, Berlin Heidelberg New York (2004) 45-60.

5. Shizuka, M., Ma, J., Lee, J., Miyoshi, Y., Takata, K.: A P2P Ubiquitous System for Testing Network Programs. In: Yang, L.T., Guo, M., Gao, G.R., Jha, N.K. (eds.): Proc. of the Conference on Embedded and Ubiquitous Computing. Lecture Notes in Computer Science, Vol. 3207, Springer-Verlag, Berlin Heidelberg New York (2004) 1004-1013.
6. Sashima, A., Izumi, N., Kurumatani, K.: Bridging Coordination Gaps between Devices, Services, and Humans in Ubiquitous computing. In: Proc. of the Workshop on Agents for Ubiquitous Computing. *http://www.ift.ulaval.ca/~mellouli/ubiagents04/*. (L.A.: July 2006)
7. Thilliez M., Delot T.: Evaluating Location Dependent Queries Using ISLANDS. In: Ramos, F.F., Unger, H., Larios, V. (eds.): Proc. of the Symposium on Advanced Distributed Systems. Lecture Notes in Computer Science, Vol. 3061, Springer-Verlag, Berlin Heidelberg New York (2004) 126-136.
8. Lemlouma, T.: Architecture de Négociation et d'Adaptation de Services Multimédia dans des Environnements Hétérogènes. PhD Thesis, Institut National Polytechnique de Grenoble, Grenoble, June 2004 (in French).
9. Carrillo-Ramos, A., Gensel, J., Villanova-Oliver, M., Martin, H.: PUMAS: a Framework based on Ubiquitous Agents for Accessing Web Information Systems through Mobile Devices. In: Haddad, H., Liebrock, L.M., Omicini, A., Wainwright, R.L. (eds.): Proc. of the 20th ACM Symposium on Applied Computing. ACM Press, New York (2005) 1003-1008.
10. Indulska, J., Robinson, R., Rakotonirainy, A., Henricksen, K.: Experiences in Using CC/PP in Context-Aware Systems. In: Chen, M.S., Chrysanthis, P.K., Sloman, M., Zaslavsky, A.B. (eds.): Proc. of the 4th International Conference on Mobile Data Management. Lecture Notes in Computer Science, Vol. 2574, Springer-Verlag, Berlin Heidelberg N.Y. (2003) 247-261.
11. Berhe, G., Brunie, L., Pierson, J.M.: Modeling Service-Based Multimedia Content Adaptation in Pervasive Computing. In: Vassiliadis, S., Gaudiot, J., Piuri, V. (eds.): Proc. of the 1st Conference on Computing Frontiers. ACM Press, New York (2004) 60-69.
12. Gandon, F., Sadeh, N.: Semantic Web Technologies to Reconcile Privacy and Context Awareness. In: Journal of Web Semantics 1(3) (2004). *http://www.websemanticsjournal.org/ps/pub/2004-17* (L.A.: August 2006).
13. Kurumatani, K.: Mass User Support by Social Coordination among Citizen in a Real Environment. In: Kurumatani, K., Chen, S., Ohuchi, A. (eds.): Proc. of the International Workshop on Multi-Agent for Mass User Support. Lecture Notes in Artificial Intelligence, Vol. 3012, Springer-Verlag, Berlin Heidelberg New York (2004) 1-16.
14. Albayrak, S., Wollny, S., Varone, N., Lommatzsch, A., Milosevic, D.: Agent Technology for Personalized Information Filtering: The PIA-System. In: Haddad, H., Liebrock, L.M., Omicini, A., Wainwright, R.L. (eds.): Proc. of the 20th ACM Symposium on Applied. ACM Press, New York (2005) 54-59.
15. Xu, J., Lim, E., Ng, W.K.: Cluster-Based Database Selection Techniques for Routing Bibliographic Queries. In: Bench-Capon, T.J.M., Soda, G., Tjoa, A.M. (eds.): Proc. of the Workshop on Database and Expert Systems Applications. Lecture Notes in Computer Science, Vol. 1677, Springer-Verlag, Berlin Heidelberg New York (1999) 100-109.

Design Options for Subscription Managers

Aloys Mbala, Lin Padgham, and Michael Winikoff

RMIT University
Melbourne, Australia
{aloys, linpa, winikoff}@cs.rmit.edu.au

Abstract. An important issue in open agent systems such as the Internet is the discovery of service providers by potential consumers (requesters). This paper is concerned with services that involve the ongoing provision of up-to-date information to requesters. We explore three separate issues: subscription to an information provider for ongoing provision of information; monitoring for new information providers; and maintaining awareness of when providers disappear from the system. We explore several models for how this functionality may best be provided, with emphasis on the ways in which certain choices affect the overall system; and provide an analysis of preferred design options for environments with different characteristics.

1 Introduction

An important issue in open agent systems such as the Internet is the discovery of service providers by potential consumers (requesters). There is a broad range of work in this area, including work on web service description languages, such as WSDL[1] and OWL-S [1], as well as work on distributed search algorithms and architectures such as peer-to-peer systems [2]. A common approach, even in peer-to-peer systems, is to have some specialized agents (or services) that assist providers and requesters to find one another. These are variously called *yellow page agents* [3], directory facilitators[2], brokers [4] and match-makers [5] with the term *middle-agent* being used to characterize these kinds of agents. UDDI (Universal Description, Discovery and Integration) directories[3] are one standard instantiation of such a facility while FIPA (Foundation for Intelligent Physical Agents) Directory Facilitators are another.

In many application areas, a large number of the services that are required from other entities in the system are services that provide information. In many cases what is required is not just information at a given point in time but rather ongoing updates of information as the situation changes. For example, in an intelligent alerting system that we are working on with the Australian Bureau of Meteorology [6], if the fire monitoring agent within the system discovers a new fire, it will then want to be informed of any weather events that may affect the fire, such as nearby storms. It is clearly preferable for the relevant agent to set up *subscriptions* and to be notified immediately when relevant

[1] http://www.w3.org/TR/wsdl
[2] http://www.fipa.org/specs/fipa00023/SC00023K.html
[3] http://www.uddi.org

M. Kolp et al. (Eds.): AOIS 2005, LNAI 3529, pp. 259–274, 2006.

new information becomes available, rather than to make regular requests to determine whether new information is available. This notion of *subscription* is well known and it is supported by standard protocols[4].

However, an additional facility is needed. If the subscriptions are long-lived then it is quite likely that there will be changes in the available information providers. The subscribing agent may well need to be made aware of new information providers that join the system, and of any information providers that it has subscribed to that leave the system. Again, rather than have the subscribing agent make periodic requests, it is preferable for it to subscribe to this information. This subscription is to changes in the available (relevant) information providers rather than to information, and is made with the middle-agent. This requires the middle-agent to provide a *monitoring* capability, in addition to the more commonly discussed *matchmaking* (or *brokering*) functionality [7].

By providing information on changes in available information providers, we allow additional flexibility and intelligence in the requesters. For example, in the meteorology application, two kinds of weather information sources are used in reasoning about whether there is an alertable situation with respect to a particular fire. If the storm observations from radar become unavailable, then storm likelihood forecasts from the atmospheric model are accessed instead. The provision of information on available relevant providers to requesters is a key difference between our work and event notification systems such as Siena [8] or NaradaBrokering [9], which do not provide requesters with information on changes to available providers[5].

In this paper, we explore design options for "Subscription Manager" middle-agents that support subscriptions to changes in available relevant information providers. There are three issues that we concentrate on. Firstly, the mechanism that allows an information requester to be continually updated regarding new information sources. Secondly, the details of how subscriptions are created and cancelled. Thirdly, how the departure of agents from the system is detected and what is done in response to detecting a "dead" agent. With each of these issues we will explore what functionality can potentially reside with the middle-agent, and the costs and benefits of the alternative approaches.

The contribution of this paper is the detailed analysis of these three issues, identifying tradeoffs and leading to recommendations regarding design choices in Subscription Manager middle-agents. We believe that these recommendations would be useful to the designer of a system that is to use a Subscription Manager middle-agent. Since our concern is with key design decisions — such as whether subscriptions should be made by the middle-agent or by requester agents — we do not provide a complete design for the Subscription Manager.

The issues discussed in this paper are only a part of a complete solution. In order to implement a system, one must also define a language for describing services and requests and a matching mechanism between these. However, these issues have been explored in previous work and a wide range of options exist for service/request description and matching including standards around web service, FIPA standards, KQML [10], and others such as LARKS [11] and Infosleuth [12].

[4] e.g. http://www.fipa.org/specs/fipa00035/

[5] What they provide corresponds to the design option where decision making is delegated to the middle-agent, i.e. what we call *subscribe-all* in Section 4.

The need for subscription and monitoring services vary from application to application, but we would suggest that they are quite broadly applicable. For example, in a travel and tourism services network, it would be likely that there was a need to subscribe to information on schedule updates for planes, buses and trains. Similarly, a tourism operator in a particular region is likely to want to monitor for any new providers of services such as accommodation, tours and car rentals in the region of interest. Similarly in an e-business domain, subscription to catalogues of items available from known providers may well make sense, and monitoring of providers of certain kinds of items is also motivated. Consequently, we argue that subscription support, and monitoring for providers of certain kinds of services joining and leaving the system, are infrastructure facilities that are required in a dynamic and open domain of services. These capabilities should be provided by middle-agents. In the rest of this paper we explore several models for how this functionality may best be provided, with emphasis on the ways in which certain choices affect the overall system.

2 The Interaction Models

Service Discovery frameworks can be categorized in two groups. The first group includes peer-to-peer dissemination models where a peer propagates its requests through the network it belongs to and expects a list of relevant providers from its peers. A peer can act as a provider, a requester or simply be a kind of proxy that just redirects a given message to others. An alternative framework uses middle-agents where requesters and providers register to a middle-agent that provides some kind of connection service to assist the agents to find other relevant agents. Some systems propose a peer-to-peer structure amongst the middle-agents [13] in order to distribute the functionality of registering and servicing the client agents.

In this work, we do not consider the structure of the middle-agents. Although we assume that in a large system this functionality would be distributed in some manner, this is left as future work, building on a range of existing work e.g. [9,8,2,13]. What we consider here is the relationship between the middle-agent (or network of middle-agents) and what we call the *end-agents*, namely the service requesters or service providers.[6]

Previous work [4,7,5,14] has compared different styles of middle-agents and concluded that *Matchmakers* that provide a list of providers matching a request are the most appropriate type of middle-agent for large open systems. Middle-agents such as broadcasters and blackboards, which simply pass on all connections, un-filtered, result in unnecessarily large lists of agents being provided and also require end-agents to have individual matchmaking capabilities. Brokers, which manage all interactions with a provider on behalf of a requester, have the disadvantage that they are a bottleneck in large systems. In this work, we assume a basic matchmaking capability and then add to this a Subscription Management function, which we explore in further detail.

There are three different processes that we explore as part of this work. The first is the mechanism to allow an information requester to be continually updated regarding the existence of new information sources of a particular kind. The second is the basic

[6] A single agent can be both a provider and a requester, but for the purpose of this work we consider them separately.

subscription mechanism to support an information requester being able to subscribe to provider agents and cancel subscriptions. The third is an ability to be aware of agents that disappear from the system. With each of these aspects, we will explore what functionality can potentially reside with the middle-agent, and the costs and benefits of the alternative approaches.

2.1 Monitoring for New Arrivals

As indicated previously, a common need in dynamic systems is for agents to be aware of new services arising in the system that may be of interest to them. One way to achieve this is to have middle-agents maintain information about requester needs and update the requesters as new providers register. However, this ability does not appear to be common in the various kinds of middle-agents that exist or are discussed in detail in the literature. Retsina [5] mentions a monitoring capability, although very little detail is given[7]. The notion of facilitator defined by Finin *et al.* [10] is broad and encompasses monitoring of both information and information providers, but little detail is given (for example, the issue of detecting "dead" agents is not discussed), and there is no exploration of the design options and associated tradeoffs.

Fig. 1 indicates the type of mechanism we are suggesting. Providers and requesters send their profiles to the middle-agent, which maintains information about both. When requesters request monitoring for a particular type of information, they are first sent an initial list of matches (message 3) and, subsequently, if any new matching providers advertise with the middle-agent (message 4), the requester is sent an update (message 5).

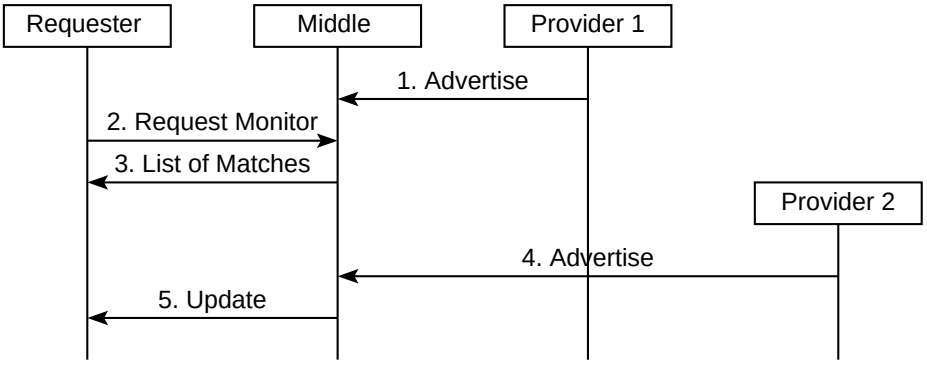

Fig. 1. The discovery mechanism

However, this figure is incomplete as it focusses only on the monitoring capability. It does not consider aspects of the subscription life-cycle such as who sets up a subscription? Who cancels a subscription? Or, once a subscription has been established, who ensures that the agents involved in the subscription are still alive? These aspects

[7] The notion of "monitor" vs. "single shot" match-making is mentioned on page 42 of [5].

are considered below. Of course, the monitoring capability must also include a mechanism for cancelling monitoring when it is no longer required or cancelling an advertised profile.

2.2 Subscription Management

In order to handle subscriptions, information providers need to be able to provide a subscription facility, sending information to their subscribers either at regular intervals or when relevant changes occur. Hence, there must be a mechanism to set up and cancel such subscriptions.

From the point of view of the information requester wishing to subscribe to a certain kind of information, they may wish to subscribe to all sources of information of a certain type, or a single source. The initial action would be a request to the middle-agent with a query describing the information need (attached to either a monitoring request or a one-off request). At that point, it would be possible either for the middle-agent to return a list of matching providers, as in Fig. 1, or for the middle-agent to simply set up the subscription(s). If the latter was done, presumably it would be necessary to have two forms of the request: one for subscribe to all and one for subscribe to one[8].

The possible value in having the middle-agent set up the subscription would be that fewer messages are needed in the system as a whole. On receiving the request, the middle-agent could simply send the subscription message to the relevant information provider(s), and the requester would begin to receive information. Subscription cancellations could be sent either to the middle-agent, or directly to the information provider, if we assume that the identity of the provider(s) is known to the requester once information begins to arrive.

2.3 Monitoring for Disappearances

If an agent has a subscription to an information source, it is expecting that information will be sent whenever relevant. However, it is possible that the information provider disappears from the system, in which case it may be important for the information subscriber to know of this. This fact may affect reasoning done or it may result in subscribing to other information sources.

For example, in the meteorology application we are working with, two kinds of weather information sources are used in reasoning about whether there is an alertable situation with respect to a particular fire. If the storm observations from radar become unavailable, then storm likelihood forecasts from the atmospheric model are accessed instead.

The only reliable way to be sure of knowing when an agent disappears is for some process to check liveness regularly. It is possible for this to be done by all interested subscribers. However, assuming there are likely to be multiple subscribers to any given information source, this is creating more message traffic than necessary. Another option would be for this to be done by the middle-agent and for the information about a provider's disappearance to be passed on to the relevant agents.

[8] An additional form would be subscribe to N.

3 Analysis

In this section, we analyse the alternative design choices for a Subscription Manager middle-agent. The analysis makes certain simplifying assumptions but is nonetheless valuable. The analysis focusses primarily on the message traffic and looks specifically at the *number* of messages, the total *size* of the messages and at *bottlenecks* in the system.

The number of messages circulating in the system is a natural and important parameter for the evaluation of service discovery frameworks since it is a reasonable approximate measure of the workload of the system, and an analysis of the message traffic received and sent by a given agent can be used to detect potential bottlenecks. However, using only the number of messages isn't sufficient because it ignores the size of the messages. Therefore we also use the size of the messages to estimate the amount of network traffic.

The analysis in this section uses the terms below. Since the analysis is done at design-time, we do not need to concern ourselves with whether the terms can be measured at run-time in a real agent system: these terms are not used at run-time.

- R: the number of requester agents in the system.
- P: the number of provider agents in the system.
- α: the probability of a random capability and a random interest matching ($0 \leq \alpha \leq 1$). This is a measure of the matching precision and can be expected to be well below 0.5.
- R_F: the (average) number of requesters whose interests match a given provider's capabilities $R_F = \alpha \times R$.
- P_F: the (average) number of providers whose capabilities match a given requester's interests $P_F = \alpha \times P$.
- S: the number of subscriptions in the system. If each requester agent subscribes to all relevant providers (P_F) then the number of subscriptions is $S = R \times P_F$. If each requester agent subscribes to P_S providers then $S = R \times P_S$.
- P_S: the (average) number of providers that a requester agent subscribes to. This can be all relevant providers (P_F), a single provider or an arbitrary number ($1 \leq P_S \leq P_F$).
- R_S: the (average) number of requesters that are subscribed to a given provider ($0 \leq R_S \leq R_F$). The value of R_S depends on whether requesters subscribe to one provider, all providers or P_S providers, and can be calculated by dividing the number of subscriptions in the system (S) by the number of providers. If each requester agent subscribes to all relevant providers (P_F) then $S = R \times P_F$ and $R_S = (R \times P_F) \div P = (R \times \alpha \times P) \div P = R \times \alpha = R_F$. If each requester agent subscribes to P_S providers then $S = R \times P_S$ and $R_S = R \times P_S \div P$, which is just P_S if there are equal numbers of providers and requesters.
- P_D: the number of provider agents that have left the system since the last liveness monitoring check ($0 \leq P_D \leq P$).
- k: the size of a description of an agent's capabilities or interests relative to the size of its name ($k > 1$). This is used in computing the size of messages.

We assign a message containing a simple request (e.g. a single name of another agent) a size of 1 and a message containing a description of the interests or capabilities of an agent a size of k. A message that contains a list has a size that is computed by multiplying the contents of the list by its length. For example, a message containing a list of P_F relevant provider names has size P_F, whereas a message containing a list of the capabilities of P_F relevant providers has size $k \times P_F$.

Our presentation of the analysis is structured according to the life-cycle of the system: we consider the metrics associated with adding an agent (requester or provider) with cancelling subscriptions and with monitoring the liveness of provider agents. In order to help make the analysis more concrete, we will include actual numbers, computed by assuming fairly arbitrary — but, we hope, reasonable — figures for the terms above. These assumed values are given in table 1, where brackets are used to indicate numbers that are derived from other values. For example, R_F is derived from R and α (since $R_F = \alpha \times R$). Two of these assumed values need explanation. Firstly, the value for P_S (and hence the value of R_S) depends on whether requester agents ask to be subscribed to one relevant provider, some constant number of relevant providers or all relevant providers. This will obviously vary depending on the requirements of the requester agents. If we arbitrarily assume that half of the requester agents ask to be subscribed to one provider, a quarter of requester agents ask to be subscribed to 5 providers and the remaining quarter of requester agents ask to be subscribed to all (in this case, on average, 10 providers), then we have that $P_S = (0.5 \times 1) + (0.25 \times 5) + (0.25 \times 10) = 4.25 \approx 4$. Secondly, the value for P_D assumes that over the course of a polling period 0.5% of provider agents will disappear. Since we have 200 provider agents, this gives one agent that will disappear in a polling period, on average.

Table 1. Example values for terms

term:	R	P	α	R_F	P_F	S	P_S	R_S	P_D
value:	200	200	0.05	(10)	(10)	(800)	≈ 4	(≈ 4)	1

3.1 Adding an Agent

Adding a Requester Agent: The sequence of messages associated with adding a requester agent depends on whether subscription is done by the middle-agent or the requester.

If subscription is done by the middle-agent then the sequence of messages is: (1) the requester registers its interests with the middle-agent, (2) the middle-agent sends messages to all relevant providers asking them to subscribe the requester, (3) the middle-agent optionally sends a message informing the requester of its subscriptions. The number of messages involved is $1 + P_F$ if the third (optional) notification message isn't sent and $2 + P_F$ if it is sent.

If we assume that each requester wants to subscribe to P_S relevant providers and that the decision of which providers can be made on its behalf by the middle-agent, then the number of messages is $1 + P_S$.

If subscription is done by the requester, then the sequence of messages is: (1) the requester registers its interests with the middle-agent, (2) the middle-agent responds with a list of relevant providers, (3) the requester selects some (P_S) or all (P_F) of the providers in the list and sends each of the selected providers a subscription request. If the requester selects a subset of the available relevant providers and the middle-agent needs to track subscriptions, then it must be notified by the requester of its choice of providers, unless it is assumed that requesters always subscribe to all relevant available providers or to some easily predicted subset such as only the first provider in the list. The number of messages involved is $2 + P_S$ (if the middle-agent needs to be informed then the number of messages goes up by one).

We now consider the message *size* and begin with the first case where subscription is done by the middle-agent. If we assume for the moment that requesters subscribe to all relevant providers (P_F), then the size of the three messages is respectively k for the first step, 1 for each of the messages involved in the second step and (optionally) P_F for the third step giving a total size of $k + P_F$ (or $k + 2P_F$ if requesters are informed of their subscriptions). If we assume that each requester subscribes to P_S providers, then the total size if $k + P_S$ (or $k + 2P_S$ if requesters are informed of their subscriptions).

Consider now the second case, where subscription is done by the requester. If we assume for the moment that requesters subscribe to all relevant providers, then the size of the three messages is respectively k, P_F, and 1 for each of the P_F messages from requester to providers, giving a total of $k+2P_F$ (and $k+3P_F$ if the middle-agent needs to be informed). If we assume that requesters will only subscribe to P_S providers, then the message to the requester containing the list of relevant providers will need to contain the provider's capabilities, as well as their names (so that the requester can decide to which providers to subscribe). Therefore, the size of the messages is $k + kP_F + P_S$ (or $k + kP_F + 2P_S$ if the middle-agent needs to be informed).

These cases are summarized in Table 2. In all cases, informing the other agent takes a single additional message of size equal to the number of desired providers. The numbers in the table give the actual number of messages, computed using the assumed values in Table 1.

Table 2. Adding a requester (message size analysis is in brackets)

	Middle Subscribes	Requester Subscribes
All providers	$1 + P_F$ ($k + P_F$) 11	$2 + P_F$ ($k + 2P_F$) 12
P_S providers	$1 + P_S$ ($k + P_S$) 5	$2 + P_S$ ($k + kP_F + P_S$) 6

In summary, having the middle-agent subscribe saves a single (potentially large) message and, if the middle-agent needs to track subscriptions, then a second message is also saved (assuming that requesters don't need to be notified of their subscriptions). However, having the middle-agent subscribe prevents a requester from being able to

directly select its provider(s) and, if requesters need to subscribe to something other than all providers, then there is additional complexity in specifying how many providers are desired (e.g. one, all or some constant number P_S).

Adding a Provider Agent: The sequence of messages associated with adding a provider agent depends on whether subscription is done by the middle-agent or the requester.

For the moment, let us assume that requesters subscribe to all relevant providers. If subscription is done by the middle-agent, then the sequence of messages is: (1) the provider registers its capabilities with the middle-agent, (2) the middle-agent sends a message back to the provider with all relevant requesters that it should subscribe (possible none) and (3) the requesters are (optionally) informed of their new subscriptions. The number of messages involved is 2 if the third (optional) notification message isn't sent and $2 + R_F$ if it is. The messages informing the requesters (step 3) could be sent by either the middle-agent or the provider. In the interests of trying to avoid overloading the middle-agent, it is preferable to have the provider inform the requesters.

If subscription is done by requesters then the sequence is: (1) the provider registers with the middle-agent, (2) the middle-agent sends a message to each relevant requester with the identity of the provider, (3) each requester sends a subscription request message to the new provider. The number of messages involved is $1 + 2R_F$. Note that there is a bottleneck issue here: the provider will, during a short time period, be sent messages from a number of requesters, potentially overloading it.

Considering the size of the messages, in the first case, where subscription is done by the middle-agent, the size of the three messages is respectively k, R_F and (optionally) 1 for each of the R_F messages giving a total size of $k + R_F$ (or $k + 2R_F$ if requesters are informed of their subscriptions). Considering the second case, where subscription is done by the requester, the size of the three messages is respectively k for the first message, 1 for each of the R_F messages, and 1 for each of the R_F messages from requesters to the provider, giving a total of $k + 2R_F$.

These cases are summarized in the top row of Table 3. Informing the requester (if the Subscription Manager subscribes) takes an additional R_F messages of size 1. The numbers in the table give the actual number of messages, computed using the assumed values in Table 1.

Table 3. Adding a provider (message size analysis is in brackets)

	Middle Subscribes	Requester Subscribes
All providers	2 ($k + R_F$)	$1 + 2R_F$ ($k + 2R_F$) 21
typical P_S providers	1 (k)	1 (k)
max. P_S providers	2 ($k + R_S$)	$1 + R_F + R_S$ ($k + R_F + R_S$) 15

The bottom two rows of Table 3 assume that requesters only want to be subscribed to a fixed number of providers. In this case when a provider joins an existing multi-agent system, most or all requesters will already have the desired number of subscriptions. This is because requesters subscribe when they join the system and departing providers are detected and replaced. Therefore, the only situation where a requester will not have its desired number of subscriptions is where there are not enough relevant providers in the system. In this case, the typical number of messages generated by a new provider joining an existing system is one (of size k) but it is possible for this to be higher: up to the (unlikely) maximum shown in the third row of Table 3. Informing the other agent takes an additional R_S messages of size 1.

In summary, if requesters subscribe to all relevant providers then having the middle-agent subscribe saves a significant number of messages and also has a saving in terms of the size of messages. Additionally, if the requesters subscribe then there are potential bottleneck issues. If requesters subscribe to a fixed number of providers then the saving is much smaller.

3.2 Cancelling Subscriptions

Cancelling a subscription can be done directly, by having the requester send a message to the provider (or vice versa if the provider is the one cancelling the subscription). Alternatively, cancelling a subscription can be done via the middle-agent. In the first case, cancelling a subscription involves a single message, with an optional second message informing the middle-agent. Both messages have size 1. In the second case, cancelling a subscription involves two messages each with size 1. Thus, the difference in terms of messages involved between direct and indirect cancellation of subscriptions is minor, and is non-existent if the middle-agent needs to be informed of the cancellation.

If a provider wishes to cancel *all* of its subscriptions, then there are a number of cases: (1) If requesters don't need to be kept informed of their subscriptions then a single message (of size 1) to the middle-agent is all that is required. (2) If requesters need to be told, but the middle-agent doesn't need to be told, then there are R_S messages from the provider to the requesters that are subscribed to it. (3) If both middle-agent and requester agents need to be informed, then there is one message from the provider to the middle-agent and R_S messages from the provider to the requesters. Although it is possible to have the middle-agent inform the requesters, this increases the load on the middle-agent, requires that the provider specify explicitly the list of subscribed requesters (unless the middle-agent has a record of subscriptions) and doesn't give any benefit.

Thus if a provider wishes to cancel all of its subscriptions, then it is most efficient to not inform the requesters but only inform the middle-agent. However, if the requesters do need to be informed then the cost of also informing the middle-agent is low.

The analysis for a requester cancelling all of its subscriptions is similar. If the requester agent does not know who it is subscribed to then it needs to first obtain the list from the middle-agent (which also has the side effect of informing the middle-agent of the cancelled subscriptions). In this case, cancelling all subscriptions requires $2 + P_S$ messages with total size $1 + 2P_S$. If the requester agent does know who it is subscribed

to, then informing the providers takes P_S messages of size 1 and informing the middle-agent is a single additional (size 1) message.

3.3 Monitoring Liveness

Providers need to be monitored, so that a provider disappearing is detected and appropriate action taken. Monitoring liveness of requesters by providers doesn't seem to make sense: if the providers have information to send, then that transmission acts as a ping[9]. If they don't have information to send, then they don't really care about the requester being alive! If monitoring of requesters is desired, then it makes sense to have the middle-agent do this.

Monitoring of providers can be done either by the middle-agent or by the requesters. Consider the first possibility. In this case, the cost for checking each provider for liveness can be worked out as follows[10]. Firstly, there are P messages to the providers. Secondly, there are P_D responses, one for each departed agent[11], where P_D is the number of departed agents found in this check (we assume that live agents do not respond). If subscriptions are done by the requester agents, then the middle-agent will need to inform the requesters ($P_D \times R_F$ messages[12]); otherwise informing the requester agents is optional.

Consider now the second possibility, where monitoring the providers is done by the requester agents. This is considerably less efficient because each provider will be monitored (redundantly!) by each requester agent that is subscribed to it. More precisely, each provider will be monitored by R_S agents. Thus $P \times R_S$ messages are sent, and $P_D \times R_S$ responses received. If the middle-agent needs to be informed, then it will (eventually) receive messages from each of the R_S requester agents that are monitoring the departed provider (an additional $R_S \times P_D$ messages).

An alternative is for the first requester agent that detects a departed provider to inform the other requester agents that are subscribed to that provider, rather than allowing them to independently realize that the provider is departed. This involves the following sequence of messages: (1) a message from a requester to the departed provider, (2) a message from the departed provider's platform to the requester, (3) a message from the requester to the middle-agent and (4) $R_S - 1$ messages from the middle-agent to the other requesters. The total number of messages for pinging a single departed provider then is $3 + (R_S - 1) = 2 + R_S$ and the message size is also $2 + R_S$. The total number of messages for pinging *all* providers is this multiplied by the number of departed providers, plus R_S messages to each live provider, i.e. $(P - P_D) \times R_S + P_D \times (2 + R_S) = P \times R_S + 2P_D$.

Note that this slightly more efficient, but more complex, approach requires that the middle-agent has a record of subscriptions (otherwise it is more expensive: replace R_S

[9] That is, we assume that the provider will detect a departed requester when it attempts to send the requester information.

[10] Note that a reasonable design decision is to spread this monitoring over a time period by gradually traversing a list of providers.

[11] The responses are sent by either the relevant agent platform (saying that the agent is unknown), or from the middleware (saying that the agent platform is unknown).

[12] If the middle-agent has an up-to-date record of the subscriptions then this can be tightened to $P_D \times R_S$.

by R_F). This approach also avoids a bottleneck issue: the middle-agent is only informed of a departed provider agent once, rather than R_S times.

A much more significant potential saving in having liveness monitoring done by requesters is that it becomes possible to exploit "implicit" pings: if a provider sends data to a requester, then this is evidence that the provider is alive and it can be assumed to have been pinged. If a provider agent is sending data frequently enough, then it will never need to be explicitly pinged as long as it is alive. If this is the case, and assuming that the optimization described above is not used, then the number of ping messages that are sent goes down from $P \times R_S$ to $P_D \times R_S$, giving $2 \times P_D \times R_S$ messages overall and $3 \times P_D \times R_S$ if the middle-agent needs to be informed. If the optimization described above is included, then the effect of implicit pings is, in the best case, to eliminate the pinging of live agents, i.e. the term $(P - P_D) \times R_S$, leaving $P_D \times (2 + R_S) = 2P_D + P_D R_S$ messages. However, it is not clear that this best case will hold, so the significant reductions promised by exploiting 'implicit' pings is perhaps exaggerated by the numbers in Table 4.

This analysis is summarized in Table 4. The bracketed formulae include informing the requesters (if the middle-agent pings) or middle-agent (if requesters ping). The third row ("Improved") is when requesters ping, but includes informing both the middle-agent and other (relevant) requester agents of a departed provider. The numbers in the table give the actual number of messages, computed using the assumed values in Table 1; the numbers in brackets include informing the requesters.

Table 4. Monitoring provider liveness (bracketed formulae include informing)

Who pings?	Number of messages	+ Implicit pings
Middle agent	$P + P_D$ $(P + P_D + P_D R_S)$ 201 (205)	N/A
Requester agents	$PR_S + P_D R_S$ $(PR_S + 2P_D R_S)$ 804 (808)	$2P_D R_S$ $(3P_D R_S)$ 8 (12)
Improved	$PR_S + 2P_D$ 802	$2P_D + P_D R_S$ 6

The analysis above only considers monitoring and detecting departed agents. What is done in response to detecting a departed agent depends on the subscription policy of the requester agents that were subscribed to the departed agent. If a requester is subscribed to all relevant providers, then there is nothing further to be done – there are no other relevant providers that could be added, because the requester is already subscribed to them. However, this doesn't mean that monitoring liveness is not important – for instance, there is a difference between receiving no information because there is no information, and receiving no information because there is no available source for the information. On the other hand, if a requester is subscribed to one provider (or, more generally, P_S providers), then a replacement provider needs to be found. How this

is done, and the number of messages involved, depends on whether subscriptions are done by the requester or by the middle-agent. The analysis is similar to that presented in Section 3.1.

4 Subscription Manager Specification

Based on the analysis in the previous section, we now specify a Subscription Manager middle-agent. The most difficult issue is regarding whether or not the Subscription manager should actually set up subscriptions on behalf of a requester. On the one hand, there is a reasonable savings in doing this and it assists with bottleneck issues at the provider. On the other hand, it removes flexibility from the requester, which may need or prefer to make its own choices. If requesters subscribe to all providers, then there is no issue with flexibility, and the savings are significant; so, in this case, it makes sense to have the Subscription Manager subscribe. On the other hand, if requesters subscribe to a fixed number of providers (and especially if this fixed number is low), then the savings are lower, and allowing the requester to select its providers becomes more important. In this case it may make more sense to have requesters subscribe themselves. Consequently, we recommend that the Subscription Manager allow *both* options.

In addition to supporting subscription being done by either requesters or the Subscription Manager, there is also a need to allow for both one-off and ongoing matching, as well as subscription to one or subscription to all[13]. This requires that the interface allows four[14] kind of requests: *single-match* (requester subscribes), *ongoing-match* (requester subscribes), *subscribe-one* (Subscription Manager subscribes the requester, and replaces if provider disappears), and *subscribe-all* (Subscription Manager subscribes requester, and subscribes to new providers as they arrive). Additionally, the Subscription Manager's interface needs to allow for a requester to cancel the ongoing-match, subscribe-one or subscribe-all, and for a provider to cancel its registration.

It is slightly more efficient for end-agents to manage cancellations directly, if the Subscription Manager does not need to be updated. If the Subscription Manager is updated, the overhead is little. Consequently, we recommend that cancellations be done directly between end-agents, since this relieves the Subscription Manager of a centralized responsibility that carries no real benefit. Requesters with an ongoing *subscribe-one* request will need to notify the Subscription Manager of the cancellation so that they can be subscribed to a new provider.

Monitoring of provider liveness can be done by either requesters or by the Subscription Manager. If we use the improved version of requester monitoring and assume that "implicit" pings completely eliminate pinging of live agents, then requester-based liveness monitoring requires fewer messages ($2P_D + P_D R_S$ compared with $P + P_D + P_D R_S$, given the assumptions of Table 1 these are respectively 6 and 205). However, this requires a more complex mechanism, shifts the responsibility for a crucial infrastructure task on to the requesters (which is not practical in an open system), and assumes that implicit pings completely eliminate pinging of live agents and that

[13] We assume that subscription to some other number must be handled by the requester.

[14] If the requester subscribes then it doesn't make sense to distinguish between subscribe-to-one and subscribe-to-all. If the middle-agent subscribes then an ongoing match is assumed.

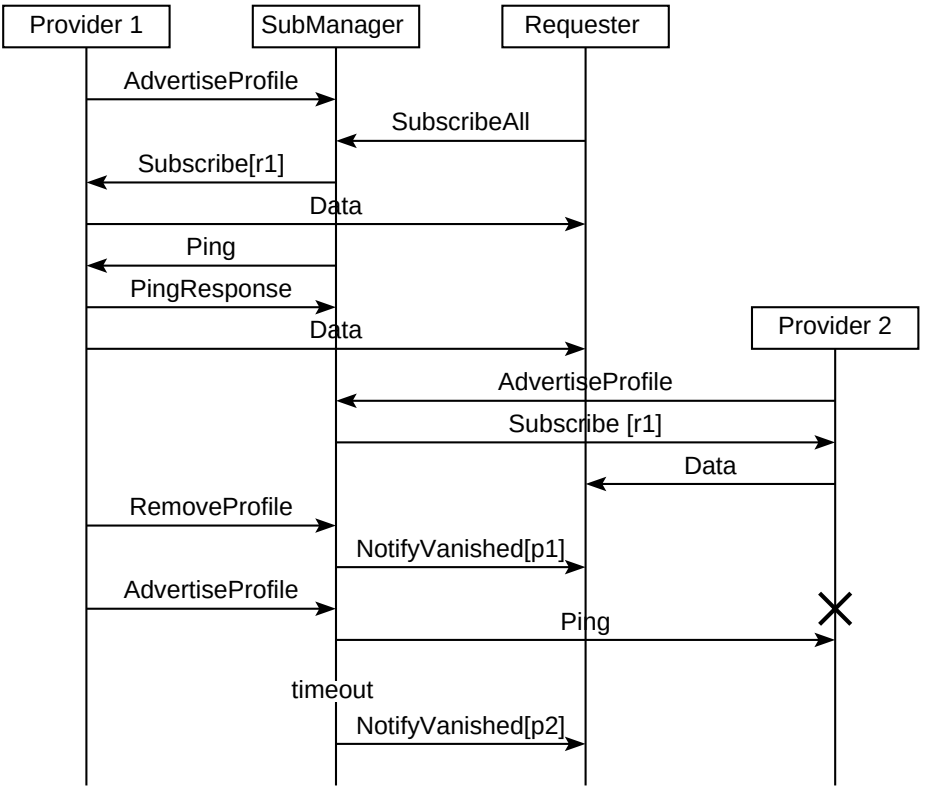

Fig. 2. An example interaction

requester agents need to be informed of departed providers[15]. Therefore, we recommend that monitoring of provider liveness be done by the Subscription Manager.

Fig. 2 shows an example interaction. In this example, a Requester has asked to be subscribed to all relevant providers (*SubscribeAll*). Provider 1 is relevant, and so the requester is subscribed (by the Subscription Manager) to Provider 1. The Provider then begins providing the requester with regular information (*Data*). The Subscription Manager also periodically checks that the provider is still available by sending *Ping* messages. A little later a second provider joins the system, and since it is also relevant to the requester, the requester is subscribed to this provider as well. The first provider then changes its service specification (*RemoveProfile* followed by a new *AdvertiseProfile*). The requester is notified that provider 1 is no longer relevant. Finally in this example, provider 2 disappears, and the Subscription Manager realizes this when it attempts to Ping the provider, at which point the requester is notified that provider 2 is no longer available.

[15] If requesters are not required to be informed of departed providers, then having middle-agents monitor providers requires $P + P_D$ messages. In this case having requesters monitor is more efficient if $P_D(1 + R_S) < P$.

5 Conclusion

We have presented a new type of middle-agent, the *Subscription Manager*, and motivated its use in systems that involve ongoing information provision to requesters. An analysis of different design options for the Subscription Manager was presented, leading to recommendations for the design of Subscription Managers. To summarize, the key recommendations are:

- that the Subscription Manager provide support for setting up subscriptions to be done either by itself or by requester agents.
- that the Subscription Manager provider a number of ways of requesting information:
  1. *single-match*, which returns a list of matching providers at the current time, but will not inform the requester of additional (relevant) providers that subsequently join the system.
  2. *ongoing-match*, which returns a list of matching providers and also asks the Subscription Manager to inform the requester should new relevant providers become available.
  3. *subscribe-one*, which asks the Subscription Manager to maintain a subscription by the requester to exactly one relevant provider (which is selected by the Subscription Manager).
  4. *subscribe-all*, which asks the Subscription Manager to maintain subscriptions by the requester to *all* relevant providers.
- that cancellations of subscriptions be done directly between end-agents.
- that monitoring of provider liveness be done by the Subscription Manager agent.

For a given application, some of the flexibility recommended may not be needed. For example, in a domain where requester agents always subscribe to all available information sources, there is no need for the Subscription Manager to support subscription to a single provider.

Areas for future work include investigating ways of structuring a *network* of middle-agents, carrying out experimental evaluation of the analysis presented and looking at how often agents should be 'pinged' given a particular rate of agent departure.

Acknowledgments

We would like to acknowledge the support of the Australian Research Council, the Australian Bureau of Meteorology and Agent Oriented Software Pty. Ltd. under grant LP0347925.

References

1. Paolucci, M., Soudry, J., Srinivasan, N., Sycara, K.: A broker for OWL-S web services. In: First International Semantic Web Services Symposium. (2004)
2. Schmidt, C., Parashar, M.: A peer-to-peer approach to web service discovery. World Wide Web Journal **7**(2) (2004) 211–229

3. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services: Concepts, Architectures and Applications. Springer-Verlag, Berlin, Germany (2004)
4. Decker, K., Sycara, K., Williamson, M.: Middle-agents for the internet. In: Fifteenth International Joint Conference on Artificial Intelligence, Morgan Kaufmann (1997) 578–583
5. Sycara, K.: Multi-agent infrastructure, agent discovery, middle agents for web services and interoperation. In: Multi-Agent Systems and Applications, LNAI 2086, Springer-Verlag (2001) 17–49
6. Mathieson, I., Dance, S., Padgham, L., Gorman, M., Winikoff, M.: An open meteorological alerting system: Issues and solutions. In Estivill-Castro, V., ed.: Proceedings of the 27th Australasian Computer Science Conference, Dunedin, New Zealand (2004) 351–358
7. Decker, K., Williamson, M., Sycara, K.: Matchmaking and brokering. In: 2nd International Conference on Multi-Agent Systems (ICMAS 1996), MIT Press (1996)
8. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Design of a scalable event notification service: Interface and architecture. Technical Report CU-CS-863-98, University of Colorado, Department of Computer Science (1998)
9. Fox, G., Pallickara, S.: The Narada event brokering system: Overview and extensions. In: Proceedings of the 2002 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02). (2002) 353–359
10. Finin, T., Fritzson, R., McKay, D., McEntire, R.: KQML as an agent communication language. In: CIKM '94: Proceedings of the third international conference on Information and knowledge management, ACM Press (1994) 456–463
11. Sycara, K., Widoff, S., Klusch, M., Lu, J.: Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace. Autonomous Agents and Multi-Agent Systems **5**(2) (2002) 173–203
12. Cassandra, A., Chandrasekara, D., Nodine, M.: Capability-based agent matchmaking. In: AGENTS '00: Proceedings of the fourth international conference on Autonomous agents, ACM Press (2000) 201–202
13. Gibbins, N., Hall, W.: Scalability issues for query routing service discovery. In: Proceedings of the 2nd International Workshop on Infrastructure for Agents, MAS, and Scalable MAS. (2001) 209–217
14. Wong, H.C., Sycara, K.: A taxonomy of middle-agents for the internet. In: 4th International Conference on Multi-Agent Systems (ICMAS 2000), IEEE Press (2000) 465–466

Supporting Program Indexing and Querying in Source Code Digital Libraries

Yuhanis Yusof and Omer F. Rana

School of Computer Science, Cardiff University, Wales, UK
{y.yusof, o.f.rana}@cs.cardiff.ac.uk

Abstract. As a greater number of software developers make their source code available, there is a need to store such open-source applications in a library and facilitate searching over this digital library. To achieve this, we propose the usage of agents in indexing and querying program source code. This paper describes agent roles in building index files for Java programs and users queries based on program structure and design patterns. Precision and recall analysis is then undertaken to evaluate the retrieval performance. We believe that such a digital library will permit better sharing of experience amongst developers and facilitate reuse of code segments.

1 Introduction

Software repositories contain a wealth of valuable information for empirical studies in software engineering: source control systems store changes to the source code as development progresses, defect tracking systems follow the resolution of software bugs, and archived communications between project personnel record the rationale for decisions throughout the lifetime of a project. Until recently, data from these repositories were used primarily for historical record — supporting activities such as retrieving old versions of the source code or examining the status of a defect. Several studies have emerged that use these data to study various aspects of software development such as software design/architecture, development process, software reuse and developer motivation.

A key motivation for our work is to facilitate software reuse through information extraction, whereby a software engineer or software developer could make use of existing software packages to create new programs. Software reuse has been shown through empirical studies to improve both the quality and productivity of software development. Our thesis is that software reuse should not just be restricted to reusing software libraries in their entirety, but should also enable software developers to understand the process associated with solving a problem encoded in the software library. A software developer may be interested in understanding how a particular feature has been coded in a particular language — rather than perhaps making full use of code that has been implemented by someone else. Despite much work in retrieving text or image documents from the Internet, less effort has been put into generating information from program

M. Kolp et al. (Eds.): AOIS 2005, LNAI 3529, pp. 275–290, 2006.

source code made available from open source projects. As the number of source code archives available on the Internet has been growing rapidly, we propose a multi-agent system for supporting program indexing and querying in source code digital libraries.

Software reuse is an approach to developing systems where artefacts that already exist are used again. Software artefacts vary from software components to analysis models. A major problem in software development today occurs when different artefacts of a software system evolve at different rates. For example, program source code is updated to include all the necessary changes, but the software models and/or formal documentations are often not modified to reflect these changes. Therefore, as the source code of systems may be the only source of information that is complete and up to date, this artefact has been widely used by software developers in program comprehension. In this paper, we concentrate exclusively on reusing such useful software artefacts — program source code. Source code can be defined as any series of statements written in some human-readable computer programming language. An important purpose of source code is for the description of software, particularly how a certain function is being undertaken. Also, source code can be used as a learning tool; beginning programmers often find it helpful to review existing source code to learn about programming techniques and methodology. It is also used as a communication tool between experienced programmers, due to its (ideally) concise and unambiguous nature. The sharing of source code between developers is frequently cited as a contributing factor to the maturation of their programming skills.

1.1 Related Work

Despite the importance of generating information from program source code, most of the research done in the area of understanding source code is focussed on categorizing the programming language used or source code achieve [1]. Ugurel *et al.* [2] classified source code into appropriate application domains and also programming languages using three components, namely the feature extractor, vectorizer and Support Vector Machine classifier. Paul and Prakash [3] have produced a framework that uses pattern languages to specify interesting code features. Therefore, a user needs to identify either the desired programming language or application domain in order to look for the desired parts of source code.

Most of the software reuse research, however, focusses on the retrieval of software components. In *Wikipedia*, a software component is defined as a system element offering a predefined service and able to communicate with other components. Ostertag *et al.* [4] classified component retrieval approaches into three types: 1) free-text keywords, 2) faceted index, and 3) semantic net based. Free-text keyword based approaches basically use information retrieval and indexing technology to automatically extract keywords from software documentation and index items with the keywords. Dongarra and Grosse [5] demonstrate the retrieval of particular numerical algorithms via email with reference to their

Netlib digital library. Many such approaches are restricted to particular types of applications (numerical algorithms in this case) and are therefore restricted in their scope. The free-text keyword approach is simple and it is an automatic process. But this approach is limited by lack of semantic information associated with keywords; thus it is not a precise approach. For faceted index approaches, experts extract keywords from program descriptions and documentation, and arrange the keywords by facets into a classification scheme, which is used as a standard descriptor for software components. To solve ambiguities, a thesaurus is derived for each facet to make sure the keyword matched can only be within the facet context. Faceted classification and retrieval has proven to be very effective in retrieving suitable components from repositories, but the approach is labour intensive. The faceted classification scheme for software reuse proposed by Prieto-Daz [6] relies on facets that are extracted by experts to describe features about components. Features serve as component descriptors, such as the components functionality, how to run the component and implementation details. To determine similarity between query and software components, a weighted conceptual graph is used to measure closeness by the conceptual distance among terms in a facet. Semantic-net based approaches usually need a large knowledge base, a natural language processor and a semantic retrieval algorithm to semantically classify and retrieve software reuse components. The semantic-net based approach is also labour intensive and often intended for use in a specific application domain. Sugumaran and Storey [7] present a semantic-based solution to component retrieval. The approach employs a domain ontology to provide semantics in refining user queries expressed in natural language and in matching between a user query and components in a reusable repository. The approach includes a natural language interface, a domain model and a reusable repository.

In motivating software component reuse, researchers have also been investigating component retrieval based on formal specifications [8,9,10,11]. Mili *et al.* [11] designed a software library in which software components are described in a formal specification: a specification is represented by a pair (S, R), where S is a set of specifications, and R is a relation on S. The approach is classified as a keyword-based retrieval system, while matching recall is enhanced with sufficient precision: a match is considered as long as a specification key can refine a search argument. There are two retrieval operations: exact and approximate retrieval. If there is no exact retrieval, approximate retrieval can give programs that need minimal modification to satisfy the specification. In measuring similarities among components, both work done by Mili [11] and Schumann [9] use automated theorem provers. Despite various techniques used in retrieving software components, there is generally no tool provided to take an existing program (i.e. written in Java) and convert it into formal specification. Existing approaches therefore require a programmer to write his/her software in a particular representation format (based on a formal specification). We see this as a severe restriction of such approaches in the context of existing source code archives.

There have been several initiatives that use agents in digital libraries, the most relevant being the University of Michigan Digital Library(UMDL) [12], The

Multimedia Electronic Documents (MeDoc) system [13] and the ZUNO Digital Library (ZUNODL) [14] — a commercial framework to build digital libraries. However, the architecture of such digital libraries is different from our approach, as virtually all of them operate on text documents. To support code retrieval, it is first necessary to remove Java language keywords, such as `println` and `bufferedreader`. Collaborative filtering may then be used to provide integration of code segments. We assume that a given source code digital library contains components written in a single programming language. Communication between agents operating on this digital library would be based on the grammar of this particular programming language. An agent may be used to inform users who have retrieved program source code from the digital library so that users are kept informed with latest information related to the retrieved program. Hence, to manage the dynamic changes in such a digital library, we propose the use of a multi-agent system, such as reported in previous work dealing with the Internet [15,16,17,12].

1.2 Our Approach

Our approach differs from existing work in that we are interested in search and retrieval techniques for program source code. We see the limited use of existing search engines for this particular problem, since search engines such as `Google.com` or `Altavista.com` only provide support for formulating a query based on keywords and phrases. The search process utilized in `SourceForge.net` makes use of keywords and is based on general descriptions given to each of the stored packages. Our intention is to extend the search process supported by such public domain software repositories and, in this work, we propose to retrieve programs based on program structure and design pattern. Such an approach is designed for developers having the intention of developing reusable software. In this work, reusable software is defined as segments of code or combinations of classes that offers code and design scavenging through concept reuse. Concept reuse, which is an alternative to software component reuse, offers a more abstract reused entity and is designed to be configured and adapted for a range of situations. Such an approach can be illustrated through various modes of problem solving, one of which can be derived by examining source code program — design patterns [18]. As design patterns abstract the key aspects of a common design structure, they overcome the inevitable constraints such as a specific interface or algorithms to be followed faced by developers in reusing software components.

Similar to indexing journal articles (author, title and year representing important features of an article), Java program structure is used to represent important features of each program stored in the program repository. We include *classes, comments, identifiers, packages and import statements* as components of the program structure. Each of these components plays a significant role in defining the functionality of a program. For example, a term *registry* identified as `class name` indicates that the instance of the program is a registry object. As it is a sound practice to name programs or function modules according to their functionality [19], meaningful identifiers are used to represent object behaviours

and attributes in a program. Using Java program structure as the basis to define our design pattern rules, we identify the general design implemented in each of the stored programs. Such an approach would benefit users in identifying the participating classes and instances, their roles and responsibilities in collaborating with each other. With a design pattern, both the problem and solution are generic enough to be independent of implementation language. Therefore, given a pertinent problem, rather than only *cutting* and *pasting* selected code segments, developers retrieving programs from our source code digital libraries are provided with design solutions relating to the problem. Hence, the same design solution can be applied in implementing similar problems that might occur in the future.

In this research, each program submitted to the digital library whether as a program to be stored or as a search query, is represented by an index file containing selected terms based on Java program structure. We then classify these programs according to the implemented design patterns. Three patterns are to be identified: Singleton, Composite and Observer. In order to search and retrieve source code from our digital library, a similarity measurement is undertaken between a users query and the index files build upon all of the stored programs. As the construction of an index file for both program submission and the user's query requires various actions, decomposing the process into smaller and more manageable chunks would be very helpful. Each of these sub-systems can then be dealt with in relative isolation. However, to present users with the optimum result, the relation between these sub-systems has to be identified. For example, the source code retrieval system might retrieve different sets of program when retrieval is carried out using program template such as in [20] or when source code retrieval is undertaken based solely on design patterns. We believe that by having different agents to extract different information, a similarity measurement between search queries and programs in the repository can be expanded, hence improving the performance of the retrieval system. Therefore, we focus on building a retrieval system that is capable of retrieving relevant program source code by cooperating results from different agents (sub-systems).

2 Agent-Based Architecture

Similar to the work done in RETSINA [15], we classify our agents into three types: *User Interface agent*, *Task agent* and *Information agent*. Agents classification depicted in Table 1 is undertaken based on the notion that interface agents are tied closely to an individual human's goals (i.e. assisting users in representing the queries). Task agents are involved in the processes associated with various problem-solving tasks (i.e. decomposing the plan (if necessary) and coordinating with other task agent or information agents for plan execution and result composition) and information agents are closely tied to data sources (i.e. retrieve required files from data source). Detailed descriptions of the agents illustrated in Table 1 are discussed further below in this section.

Table 1. Classification of Agents

User Interface Agent(UIA)	Information Agent(IA)	Task Agent(TA)
Program Representation Agent(PRA)	Registry Agent(RA)	Index Builder Agent(IBA)
Index Representation Agent(IRA)	Program Agent(PA)	Index Creation Agent(ICA)
Query Representation Agent(QRA)		Program Management Agent(PMA)
Report Representation Agent(RRA)		Stemming Agent(STEMA)

The software digital library is based on the cooperation of the interface, task and information agents. In Fig. 1, we illustrate the general multi-agent system architecture for program indexing and querying in our source code digital library. As the main focus of our system is to retrieve and index Java programs, currently we are using three task agents in supporting the process of creating suitable metadata for Java programs. Upon combining all indices generated by these agents, the index file will be created and stored in the registry by the information agents. We then use this file as the main source of our comparison mechanism.

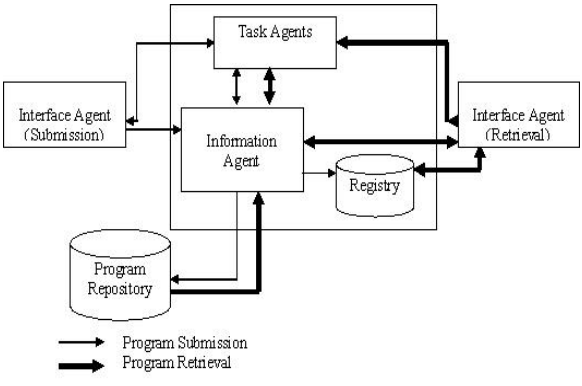

Fig. 1. The Proposed Multi-agent System Architecture for Program Indexing and Querying

The User Interface Agent (UIA) has two different roles in this architecture. From the developers view, it is responsible for accepting programs or a project folder to be submitted to the program repository. A project folder may contain a number of Java programs organized as a Java package or may just include a single Java file. Nevertheless, if a developer is using the system to retrieve program source code, UIA will act as a medium to accept search queries. All of the submitted input received by UIA are given an `ID` to differentiate whether it is a program submission or a search query. This `ID` is important in order to determine if the index to be generated should be stored in the index registry or only used by Program Matcher Agent(PMA). Users who intend to retrieve programs from the software digital library are given the flexibility of submitting two types of queries: phrase **(Query 1)** and program **(Query 2)**. Examples of these are provided below:

Query 1:

```
registry class implementing Singleton
```

Query 2:

```
public class Registry {
private static Registry registry = null;
private static final Object classlock = Registry.class;
private int connectionCount;
private Registry (){ }
public void addToCount() {
connectionCount++; }
public static Registry getRegistry() {
synchronized(classlock){
if (registry == null) {
registry = new Registry(); }
return registry;
} } }
```

The Program Representation Agent (PRA) represents any document posted by the programmer — for instance a folder containing several Java program files or a single Java program.

The Registry Agent (RA) is an information agent responsible for managing the index registry. All program index files (including programs full path name) are stored in this registry and these files are used during the matching process. The Program Agent (PA) manages the program repository by storing and retrieving the required Java files, for example retrieving the selected program source code as required by users.

The most important agent in this program source code retrieval system is The Index Creation Agent (ICA) which consists of three agents: a Keyword agent (KEMA), a Design Pattern agent (DEPA) and a Java Template agent (TEMA).

- Similar to text mining, in KEMA each word of the Java program will be analysed separately as an individual token. A complete lexicon of terms excluding those terms defined in the stop list will be undertaken. The stop list contains all words that do not provide any meaningful information in the retrieval process such as *the, a, an, void, and etc.*. Upon removing these words, the selected words (after word stemming [21] by STEMA) may then be used as the metadata for the particular Java file. However, only identifiers written as `class`, `package` or `method` name and words in `comment statements` are included in the analysis. An index built based on the processed Java file will then consists of: (1) term, (2) type of the term, and (3) absolute pathname Based on Query 2, three terms are extracted from the query program: *registry, addtocount* and *getregistry*. Therefore the index file for the query contains the following:

C:\project1registry.java
registry, class
addtocount, method
getregistry, method

- TEMA is responsible for extracting information based on program structure [20]: (1) class name(s), (2) absolute pathname, (3) method name and signatures, (4) superclass, (5) abstract class, (6) interface class. This information is then used to generate alternative indices, which can be used to retrieve similar programs based on the search query. Example of indices generated by this agent based on Query 2 are as following:

  C:\project1registry.java
  method addtocount - parameters: null; return: null
  getregistry - parameters:null; return: registry

  Method named *addtocount* as illustrated, does not receive any argument as well as not returning any data. Hence, to provide flexibility in matching search queries and programs in the repository, TEMA generates alternative data such as providing data type *integer* as the method parameter. Such an approach would benefit users in retrieving similar programs since program matching is not bound to the exact program structure.
- DEPA is accountable for identifying three design patterns that are implemented in a Java program. This agent determines the existence of design patterns based on several rules. The outcome of this identification is the percentage of rules obeyed in determining the design patterns. To summarize, a program is identified to be implementing a Singleton if: (1) it only allows a single creation of an instance and (2) access to private class variables is implemented in a public method. To identify the existence of a Composite design pattern, DEPA: (1) identifies classes implementing at least one interface and (2) determines whether the identified classes provide a method that receives an interface class as its argument. For a class to be identified as implementing the Observer pattern, it must have the following: (1) private variable(s) which allows the value that it holds to be updated, (2) inheritance of any abstract classes — where an abstract class defines the identity of its descendants, (3) method overriding between a class and its superclass (abstract class) and (4) a constructor that receives at least one element from rule (1) as its method argument.

These three agents cooperate between each other in order to fulfil each others goals. For example, given a Java program as the search query, QRA will invoke TEMA to analyse the Java program. For TEMA to produce its template indices, it requests KEMA to identify lexical terms. With these terms, TEMA will then generate further indices for the query program. DEPA is then invoked to determine the existence of any defined design patterns.

The Index Builder Agent (IBA) combines indices generated by ICA. Based on the data received from ICA, it generates a general index to represent each of the

processed Java file. The data structure of the index consist of: (1) the absolute pathname; (2) a vector of objects containing terms and type of the terms. Types of terms are determined based on the program structure - class name, method name, package name and comments; (3) a vector of objects storing particulars about a class - class name, superclass name, method signatures, abstract class and interface class(s); (4) the percentage of the existence of design patterns.

The Index Representation Agent (IRA) is responsible for indices generated by the IBA to be presented to the user. It creates a report containing all of the generated indices from the particular program file. This report is then presented to the person who submits the program. Based on the example in Query 2, the generated report contains the following:

File name : C:\project1registry.java
Class name = registry
No.of selected terms = 3
Design Pattern= Singleton(100%)

The Query Representation Agent (QRA) is responsible for formatting users queries into an appropriate form. For example, if a user submits a folder of Java programs, QRA allows users to specify their query using two different modes: a description of what they are searching for in human language (English) and a Java template of the query.

The Program Matcher Agent (PMA) is in charge of finding suitable Java programs based on users queries. The similarity comparison is undertaken between two index files: a query index file and an index file for all programs stored in a registry. Indices in the query index file are mapped against all indices in the registry index file using two similarity measurements: string and design patterns. String and substring matching is undertaken based on the Levenshtein distance function [21]. A threshold value is to be requested from the user in order for PMA to find similar Java programs that contain terms that produce the minimum value of the distance function. Similarities in terms of design patterns is obtained by comparing the percentage value of the existence of design patterns contained in both of the programs index file and query index file. Selected Java files are then ranked according to their sum of distance function values and percentage of design pattern existence. The program with the lowest and highest value, respectively, are presented as the most suitable programs.

Upon having a list of relevant Java files (undertaken by the PMA), related Java files references are passed to the Result Representation Agent (RRA). This agent plays the role of presenting the results to the users by generating a report describing the selected Java programs. This report nevertheless contain information on similar terms and design patterns found between the search query and the program. It is also responsible for fetching any selected programs(from the result report) required by the users. This is achieved with the cooperation of the program agent(PA).

The majority of interactions of interface agents are with the human user, the most frequent interactions of information agents are with information sources, whereas task agents interact with other task and information agents. When a

task agent receives a task from an interface agent, or from another task agent, it decomposes the task based on the domain knowledge it has and then delegates the sub-tasks to other task agents or directly to information agents. The task agent will take responsibility for collecting data, coordinating among the related agents (i.e. agents which accept the sub-task) and report back to whomever initiated the task. The agent who is responsible for the assigned sub-tasks will either decompose these sub-tasks further or perform data retrieval. Upon receiving a task, an agent starts planning using its own operator and behaviour. If it requires other operator that does not exist in its domain knowledge, it must find and request another agent that has the capabilities to complete the remainder of the task. This process will continue until an agent can achieve the goal of the received request by itself. In Fig. 2, we illustrate a portion of the plan library containing general descriptions of action decomposition methods expressed in the form `Decompose(a,b)`. This says that an action `a` can be decomposed into the plan `b`, which is represented as a partial-order plan.

```
Action(BuildTerm, PRECOND:Program,
EFFECT:TermList)
Action(BuildDP, PRECOND: Singleton ∧ Composite
∧ Observer, EFFECT:DPList)
Action(Construction, PRECOND:TermList ∧
DPList, EFFECT:Index)

Decompose(BuildIndex,
Plan(Steps:{S1:BuildTerm, S2: BuildDP,
S3:Construction}
Orderings:{Start ≺ S1 ≺ S3 ≺ Finish, Start ≺
S2 ≺ S3 }
Links:{Start ⟶ S1, Start ⟶ S2, S1 ⟶ S3,
S2 ⟶ S3, S3 ⟶ Finish})
```

Fig. 2. Action Descriptions for the Index Building of a Program

3 Agent Interaction for Program Submission and Program Retrieval

We classify our users into two categories: developers who intend to submit their Java application into the repository and users who requires Java programs from the repository.

3.1 Program Submission

In Fig. 3, we depict communication that occurred between agents during the process of program submission through a sequence diagram. Using the same program illustrated in Query 2, we demonstrate how the index file for a program is generated and stored in the repository.

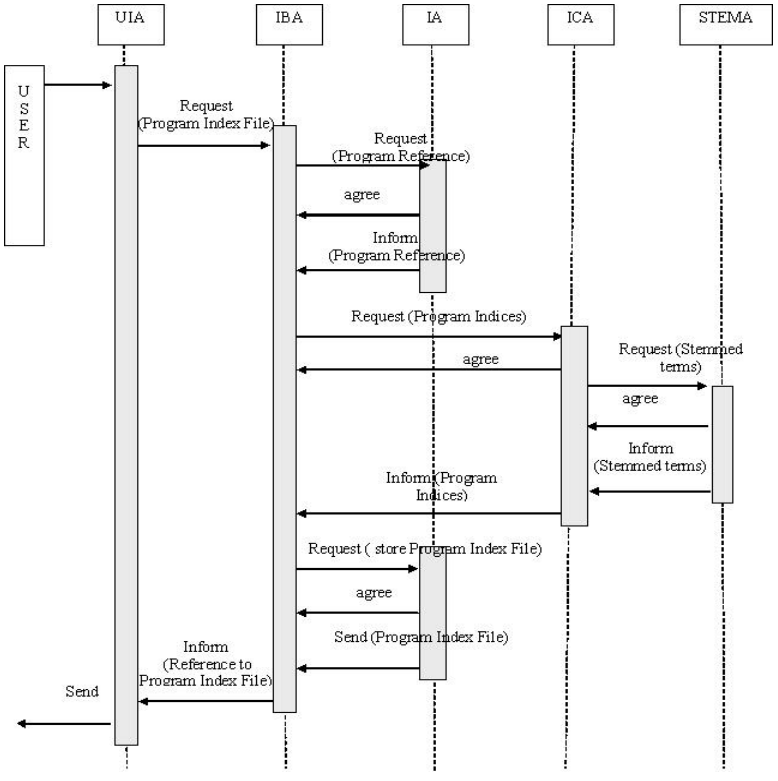

Fig. 3. Sequence Diagram for Program Submission to Repository

UIA sends a request to IBA to build an index file for *Registry.java*. Upon receiving this request, IBA notifies the message sender using one of the four ACL performatives: `agree, refuse, not understood` or `failure`. If IBA agrees to build an index file for *Registry.java*, it requests PA (responsible for managing program repository) of type information agent, to check whether *Registry.java* exists in the repository. If *Registry.java* has been submitted to the system, PA then informs IBA of the current address of the related project, or else it stores *Registry.java* in the repository and informs the reference address to IBA. Upon receiving this reference, IBA sends a `request` performative to all Index Creation Agents (ICA) asking each of them to build indices for the given reference (*Registry.java*). If these agents agree to perform the task, each of the ICAs request STEMA to perform stemming towards the Java file. They will then inform IBA about the indices that they have generated after analyzing *Registry.java*. Upon receiving these indices, IBA builts an index file to represent *Registry.java* — all of the indices are combined into a single data structure and passed to RA (responsible for managing registry) of type information agent. This agent then updates the index registry with the file that it has just received. IBA then returns back to UIA, providing a Web reference for the generated index file.

3.2 Program Retrieval

Two different modes of queries are currently supported: (i) keyword or phrase (natural language) describing user requirements and (ii) Java program or template, In Fig. 4, we describe how agents perform their task of retrieving relevant Java programs based on a text phrase as the search query.

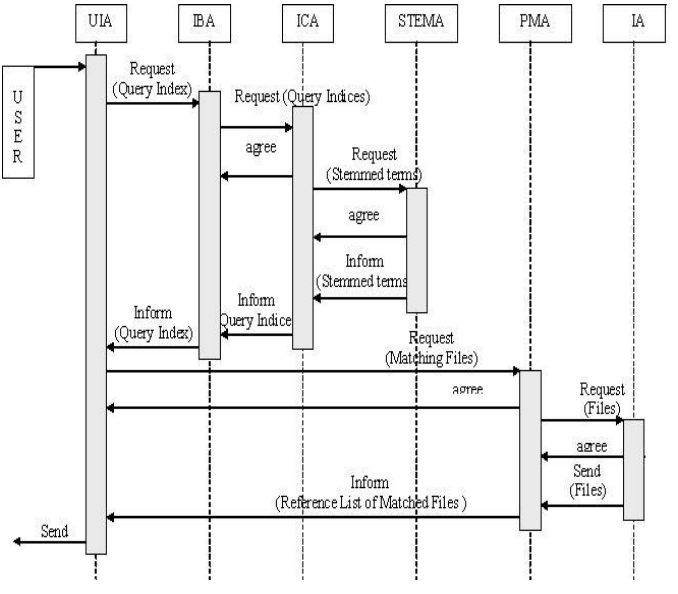

Fig. 4. Sequence Diagram for Program Retrieval from Repository

If a user submits a description of their search requirements as follows: *registry class implementing Singleton*, the UIA requests the IBA to build an index to represent this query. To perform this task, the agent IBA requests the ICA to build indices for the query. In order to do this, ICA requests STEMA to perform stemming towards the query. If STEMA understood the message and agrees to perform the task, it sends a message back to ICA containing the result. As a text query only involves lexicon terms, ICA then passes these terms to IBA in order for IBA to generate an index for the users' query (query index). Upon receiving the query index file from IBA, UIA then requests PMA to deliver Java programs that are relevant to the users query. To fulfil this task, PMA requests the information agent (RA) to sequentially retrieve index files stored in the program registry. It is then PMA's responsible to perform similarity measurements between the query index and program index files and store the reference(s) for the matched Java file(s). Upon completing the search, PMA informs UIA of the list of relevant files.

On the other hand, if a user employ a Java program as his/her search query, all of the ICA (KEMA, DEPA and TEMA) are invoked to evaluate the program.

Before this happens, UIA requests IBA to build an index file for the query. To fulfil this task, IBA then requests all of ICA to generate indices for the submitted program. Upon sending an `agreement` performative to IBA, each of the ICA requests STEMA to stem the content of the Java file. As IBA receives results from the requested agents, it combines the generated indices into one query index file. The UIA then requests PMA to search for Java programs that are similar to the search query, and this process continues as described in the above paragraph.

4 Case Study and Discussion of Results

Similar to search engines, our program source code retrieval system returns a list of documents (the hitlist) for a query. Typically, there are some good documents in the list and some bad ones. The quality of a search retrieval system is measured in terms of the proportion of good hits in the list, the positions of good hits relative to bad ones, and the proportion of good documents missing from the list. To illustrate how program structure and concept reuse can be adopted in source code retrieval system, we perform a relevant experiment using 7 applications (477 files) representing the mathematical domain, obtained from the `SourceForge.net` repository. Queries posted to the retrieval system mainly involve programs that exemplify the three identified design patterns - Singleton, Composite and Observer. In order to evaluate the retrieval performance, prior to the experiment, we manually identify relevant Java files to be used as the retrieval answer set.

Table 2. Precision and Recall Analysis

Design Patterns	Precision	Recall
Singleton	4%	100%
Composite	56%	37.84%
Observer	88%	81.48%

In Table 2, we summarize the traditional measures of retrieval performance: Recall and Precision [21] based on the top 50 Java files included in the hitlist by our retrieval system. We learn that the system has performed well by retrieving 74 out of 130 relevant files, which results in a 56.92% success rate. Both detection of programs illustrating Composite and Observer design patterns produced more than 50% precision. On the other hand, 4% has been obtained for retrieving programs implementing the Singleton design pattern. Nevertheless, such results can be improved by manipulating the retrieval cut-off point. For example, if we change the default retrieval cut-off point from 50 to 10, the precision of retrieving files implementing the Singleton design pattern is increased from 4% to 10%.

A better description of the source code retrieval performance is given based on the precision and recall graph as depicted in Fig. 5. Successfully, our retrieval system has been able to achieve a complete recall in retrieving programs that

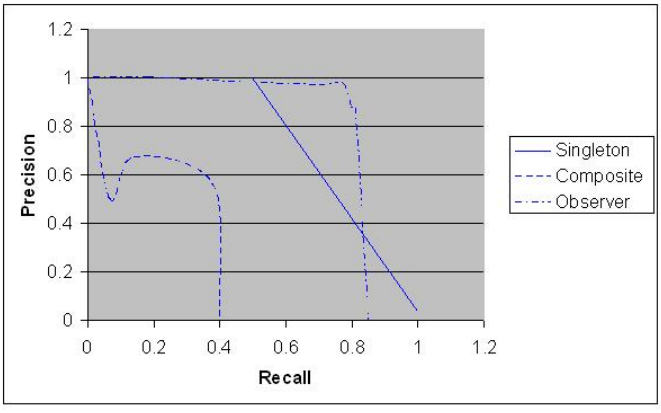

Fig. 5. Precision vs. Recall

illustrates the use of the Singleton design pattern. Based on the retrieval answer set, the first Java file in the retrieval hitlist generated a precision of 100% at 50% recall while the second Singleton file which was positioned as the 50th file in the hitlist produced a precision of 4% at 100% recall. In retrieving Java files that are relevant to implementing the Composite design pattern, we learn that the first file in the hitlist is ranked as the top file in the Composite retrieval answer set, hence producing 1 as the precision value. The retrieval precision value dropped before it raised to the value of 0.67 when we examined the 10th retrieved file. However, this value is then permanently decreased as we evaluate more relevant files in the retrieval hitlist, hence producing an increment in recall. The recall value obtained for the 50th Java file in the hitlist is approximately one third of the complete recall while its precision remained at roughly 0.5. Based on the graph depicted in Fig. 5, using 50 as the retrieval cut-off point, the precision at levels of recall higher than 0.4 dropped to 0 because not all relevant files have been retrieved. On the other hand, the optimal result throughout this code retrieval experiment is obtained through the trial of retrieving Java files that demonstrate the Observer design pattern. The retrieval system performed convincingly by retrieving almost 82% of files in the answer set, hence generating only a small portion of relevant Java files missing from the list. Even though, as illustrated in Fig. 5, the system did not obtain a complete recall for both Observer and Composite design pattern, we believe that such a goal can be achieved by increasing the retrieval cut-off point. Nevertheless, the retrieval system might be returning more non relevant files.

Even though we did not achieve complete recall for all queries submitted to the system, we illustrate that our retrieval system is capable of motivating developers in software reuse particularly in concept reuse. This is attained by configuring our matching mechanism to present users with Java programs that

not only illustrate an identical match between query programs and programs in the repository but also to retrieve files that exemplify portions of design patterns as occurred in the query programs.

5 Conclusion

With the emerging interest in making source code available, and the significant emphasis being placed on this by many software architects, digital libraries that support the searching of source code have become necessary. We show that program indexing can improve scientific communication by revealing hidden knowledge such as design patterns in programs. By utilizing a multi-agent system where all agents undertake specific roles within the system, we facilitate the process of indexing and searching Java source code in a source code digital library. As demonstrated, using agent technology we not only can increase the percentage of retrieving relevant documents in a source code digital library but also assist developers in identifying general designs that address a recurring design problem in object-oriented systems. As most of the programmers and developers learn by studying available code, being presented by various programs (which are relevant to the queries) is believed to motivate code and concept reuse.

References

1. Ruben, P., Peter, F.: Classifying software reuse. IEEE Software **4**(1) (1987) 616
2. Ugurel, S., Krovetz, R., Giles, C.L.: What's the code?: automatic classification of source code archives. In: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining, ACM Press (2002) 632–638
3. Santanul, P., Atul, P.: A framework for source code search using program patterns. IEEE Transaction on Software Engineering **20**(6) (1994) 463–475
4. Ostertag, E., Hendler, J., Daz, R.P., Braun, C.: Computing similarity in a reuse system: An al-based approach. ACM Transactions on Software Engineering and Methodology (TOSEM) **1**(3) (1992) 205–228
5. Dongarra, J.J., Grosse, E.: Distribution of mathematical software via electronic mail. Communications of the ACM **30**(5) (1987) 403–407
6. Prieto-Diaz, R.: A Software Classification Scheme. Phd thesis, Department of Information and Computer Science, University of California (1985)
7. Sugumaran, V., Storey, V.C.: A semantic-based approach to component retrieval. The DATA BASE for Advances in Information Systems **34**(3) (2003) 8–24
8. Penix, J., Alexander, P.: Using formal specifications for component retrieval and reuse. In: Proceedings of the 31st Hawaii International Conference on System Sciences. (1998) 356–365
9. Schumann, J., Fischer, B.: Nora/hammr: Making deduction-based software component retrieval practical. In: Proceedings of the 1997 International Conference on Automated Software Engineering(ASE'97), Lake Tahoe, CA (1997) 246–254
10. Nakkrasae, S., Sophatsathit, P.: A formal approach for specification and classification of software components. In: Proceedings of the 14th international conference on Software engineering and knowledge engineering, ACM Press, New York (2002) 773–780

11. Mili, A., Mili, R., Mittermeir, R.T.: Storing and retrieving software components: A refinement based system. IEEE Transactions on Software Engineering **23**(7) (1994) 445–460
12. Birmingham, W.P., Durfee, E.H., Mullen, T., Wellman, M.P.: The distributed agent architecture of the university of michigan digital library (extended abstract). In: (AAAI) Spring Symposium on Information Gathering. (1995)
13. Barth, A., Breu, M., Endres, A., de Kemp, A., eds.: Digital Libraries in Computer Science: The MeDoc Approach. Springer-Verlag Heidelberg (1998)
14. Derbyshire, D., Ferguson, I.A., Muller, J.P., Pischel, M., Wooldridge, M.: Agent-based digital libraries: Driving the information economy. In: Proceedings of the Sixth IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises. (1997) 82–86
15. Sycara, K., Decker, K., Pannu, A., Williamson, M., Zeng, D.: Distributed intelligent agents. IEEE Expert **11**(6) (1996) 36–46
16. Linn, C.N.: A multi-agent system for cooperative document indexing and query in distributed networked environments. In: Proceedings of the International Workshop on Parallel Processing, Japan (1999) 400–405
17. Kusumura, Y., Hijikata, Y., Nishida, S.: Text mining agent for net auction. In: ACM Symposium on Applied Computing, Nicosia, Cyprus (2004) 1095–1102
18. Sommerville, I.: Software Engineering. 7th edn. Addison-Wesley (2004)
19. Rodriguez, H.: Good programming practice. (`http://www.start-linux.com/articles/article_75.php`)
20. Yusof, Y., Rana, O.F.: Template mining in source code digital libraries. In: Proceedings of the International Workshop on Mining Software Repositories, 26th International Conference on Software Engineering, Edinburgh, UK (2004) 122–126
21. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison Wesley (1999).

Author Index